



Meta-Stochastic Simulation for Systems and Synthetic Biology using Classification

Daven Sanassy BSc (Hons)

Thesis submitted for the degree of Doctor of Philosophy

July 2015

Declaration of Authorship

I, Daven Sanassy, declare that this thesis titled, ‘Meta-Stochastic Simulation for Systems and Synthetic Biology using Classification’ and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

“Biology has become a sort of branch of computer science, genes are just long computer tapes, and they use a code which is just another kind of computer code, it’s quaternary rather than binary but it’s read in a sequential way just like a computer tape, it’s transcribed, it’s copied and pasted. All the familiar metaphors from computer science fit. This is a complete turnabout from the way biology used to be, where one talked in terms of a vital fluid. Now we’re becoming wholly mechanistic when talking about life. It’s a great revelation to all of science. It’s a most thrilling and exciting time for a scientist to be alive.”

Richard Dawkins

Abstract

To comprehend the immense complexity that drives biological systems, it is necessary to generate hypotheses of system behaviour. This is because one can observe the results of a biological process and have knowledge of the molecular/genetic components, but not directly witness biochemical interaction mechanisms. Hypotheses can be tested *in silico* which is considerably cheaper and faster than “wet” lab trial-and-error experimentation. Bio-systems are traditionally modelled using ordinary differential equations (ODEs). ODEs are generally suitable for the approximation of a (test tube sized) *in vitro* system trajectory, but cannot account for inherent system noise or discrete event behaviour. Most *in vivo* biochemical interactions occur within small spatially compartmentalised units commonly known as *cells*, which are prone to stochastic noise due to relatively low intracellular molecular populations.

Stochastic simulation algorithms (SSAs) provide an *exact* mechanistic account of the temporal evolution of a bio-system, and can account for noise and discrete cellular transcription and signalling behaviour. Whilst this reaction-by-reaction account of system trajectory elucidates biological mechanisms more comprehensively than ODE execution, it comes at increased computational expense. Scaling to the demands of modern biology requires ever larger and more detailed models to be executed. Scientists evaluating and engineering tissue-scale and bacterial colony sized bio-systems can be limited by the tractability of their computational hypothesis testing techniques.

This thesis evaluates a hypothesised relationship between SSA computational performance and biochemical model characteristics. This relationship leads to the possibility of predicting the fastest SSA for an arbitrary model - a method that can provide computational headroom for more complex models to be executed. The research output of this thesis is realised as a software package for meta-stochastic simulation called *ssapredict*. *Ssapredict* uses statistical classification to predict SSA performance, and also provides high performance stochastic simulation implementations to the wider community.

Acknowledgements

I wish to thank my supervisor, Natalio Krasnogor, for bestowing his guidance, vision, knowledge and timely praise upon me. Any conversation with Natalio is eye-opening and stimulating as he passes on his infectious passion for the scientific state-of-the-art (over multiple disciplines), and the creative possibilities therein. I am sincerely grateful for the opportunity that I have had to study under his tutelage.

Thanks to senior colleagues who took the time to pass on their knowledge and advice; from whom I have learnt a great deal: Paweł Widera, Harold Fellermann, Jonathan Blakes & Jamie Twycross. Thank you to my close colleagues Jurek Kozyra, Nunzia Lopiccolo, Charles Winterhalter, Maria Franco and others from the Interdisciplinary Computing and Complex BioSystems (ICOS) research group. Thanks also to project colleagues from other institutions: Steven Higgins, Christophe Ladroue, Savas Konur & Felix Dafhnis-Calas.

Special thanks to my wife Rouku and my parents for their unwavering support, encouragement and affection. And to my dearest friends Adam Miller (1982-2014) and David Clayton for supporting me and inspiring me to achieve.

Finally, I wish to thank the institutions that have supported me financially through my PhD studies, Newcastle University & University of Nottingham Computing Science departments. And the *Engineering and Physical Sciences Research Council (EPSRC)* for funding two synthetic biology projects I have been involved with:

- *ROADBLOCK: Towards Programmable Defensive Bacterial Coatings & Skins*
[EP/I031642/1]
- *AUDACIOUS: Towards a Universal Biological-Cell Operating System*
[EP/J004111/1]

Contents

Declaration of Authorship	iii
Abstract	v
Acknowledgements	vi
1 Introduction	1
1.1 Background and motivation	1
1.2 Aims and scope	4
1.3 Structure of thesis	6
1.4 Main contributions	7
1.4.1 <i>ngss</i> : Next Generation Stochastic Simulator	7
1.4.2 <i>ssapredict</i>	8
1.4.3 <i>SSA benchmarking suite</i>	9
1.5 Published work	10
2 Background Theory	13
2.1 Introduction	13
2.2 Biological overview	13
2.3 Biochemical modelling	16
2.4 Stochastic Simulation Algorithms (SSA)	18
2.4.1 Introduction	18
2.4.2 First Reaction Method & Direct Method	19

2.4.3	Worked through example of Direct Method for a simplified network	22
2.4.4	Next Reaction Method	24
2.4.5	Optimised Direct Method	26
2.4.6	Sorting Direct Method	28
2.4.7	Logarithmic Direct Method	29
2.4.8	Partial Propensity Direct Method	31
2.4.9	Composition Rejection	33
2.4.10	Tau Leaping	36
2.4.11	Reaction dependency graph (RDG)	39
2.5	Modelling genetic biochemical systems	41
2.5.1	Modelling synthetic genetic logic gates	41
2.5.1.1	Systems Biology Markup Language (SBML)	44
2.5.2	Simulating synthetic biological boolean gates	45
2.5.3	Benchmarking models	46
3	Modelling and Stochastic Simulation of Biochemical Models	51
3.1	Introduction	51
3.2	Simulating models from the literature	52
3.2.1	Experimental models	52
3.2.2	Model A1: Robust cAMP oscillations during <i>Dictyostelium</i> aggregation	52
3.2.3	Model A2: Heat shock response in <i>Escherichia coli</i>	56
3.2.4	Model A3: <i>Escherichia coli</i> AI-2 quorum sensing circuit	58
3.2.5	Model A4: Thermodynamic switch modulating abscisic acid receptor sensitivity	60
3.2.6	Model A5: Auxin transport case study	62
3.2.7	Model A6: G Protein Signalling	65
3.2.8	Model A7: Discrete proliferation model	67
3.2.9	Model A8: Stochastic model of lacZ lacY gene expression	70
3.3	Summary & conclusions	72

4	Characterising Biochemical Models	75
4.1	Introduction	75
4.2	Computing properties of biochemical models	76
4.2.1	Biochemical models as graphs	76
4.2.2	Experimental models	77
4.2.3	Graphs and graph theory	78
4.2.4	Exemplar reaction network & reaction dependency graph (RDG) generation	80
4.2.5	Species network graph (SNG) generation	82
4.3	Analysis of graphs	83
4.3.1	Number of graph vertices and edges	83
4.3.2	Graph density	84
4.3.3	Graph degree	85
4.3.4	Weakly connected components	85
4.3.5	Articulation points	86
4.3.6	Biconnected components	86
4.3.7	Directed graph reciprocity	87
4.3.8	Shortest paths in graph	87
4.3.9	Centrality	88
4.3.9.1	Closeness centrality	88
4.3.9.2	Betweenness centrality	89
4.3.10	Average geodesic length	90
4.3.11	Girth of graph	90
4.3.12	Graph transitivity	91
4.3.13	Connectivity	92
4.4	Model property analysis	93
4.4.1	Methods	93
4.4.2	BioModel property correlation analysis	93
4.4.3	Dataset comparison and analysis using model topological prop- erties	97
4.5	Property computability and complexity	102

4.5.1	$O(M)$ Reaction dependency graph generation	103
4.6	Summary & conclusions	104
5	Benchmarking Stochastic Simulation Algorithms	107
5.1	Introduction	107
5.2	Benchmarking suite	108
5.2.1	Overview	108
5.2.2	Algorithm implementations	109
5.2.3	Benchmark models	110
5.3	Simulation algorithm accuracy testing	111
5.4	Measuring algorithm performance	114
5.5	Preliminary BioModels performance analysis	116
5.6	Quantitative BioModels performance analysis	123
5.7	Summary & conclusions	126
6	Principled Selection of Stochastic Simulation Algorithm	129
6.1	Introduction	129
6.2	Experimental roadmap	130
6.3	SSA performance estimation using linear regression	132
6.4	Feature selection for SSA performance estimation	133
6.5	SSA performance classifier experiments: Methods	135
6.6	SSA performance classifier experiments: Results	136
6.6.1	Cross-validation experiments	136
6.6.2	Real world models experiments	139
6.6.3	Prediction results summary	141
6.7	SSA performance classifier experiments: Assessing mispredictions	142
6.8	SSA performance classifier experiments: Large scale models	144
6.9	Summary & conclusions	146
7	Ssapredict: Meta-Stochastic Simulation Web application	151
7.1	Introduction	151
7.2	Web application overview	152

7.3	<i>Ssapredict</i> walk-through	153
7.4	Model property generation	156
7.5	<i>Ssapredict</i> (fastest SSA) predictor	157
7.6	Software engineering	158
7.6.1	<i>Ssapredict</i> web application	158
7.6.2	<i>Propertygen</i> SBML model graph property generator	160
7.6.3	Linear SVC model-algorithm performance predictor	161
8	Next Generation Stochastic Simulator (<i>ngss</i>)	163
8.1	Introduction	163
8.2	Simulating models With <i>ngss</i>	164
8.3	Simulation parameters	165
8.3.1	Available simulation parameters & types	165
8.4	Software engineering	167
8.4.1	External software libraries	170
8.4.1.1	Boost	171
8.4.1.2	GSL	171
8.4.1.3	libSBML	172
8.4.1.4	HDF5	173
8.4.1.5	RapidXml	174
8.4.2	Parallelising stochastic runs	174
8.4.2.1	OpenMP	175
8.4.2.2	OpenMPI	176
9	Conclusions	179
9.1	Summary of thesis motivation	179
9.2	Evaluation of hypotheses	180
9.3	Knowledge transfer	181
9.4	Limitations & reflections	182
9.5	Future research directions	183
9.5.1	Online meta-SSA	183
9.5.2	Increasing SSA adoption	183

9.5.3	Scaling up the SSA	184
A	Statistical Methods	185
A.1	Pearson product moment correlation coefficient	185
A.2	Mann-Whitney U test	186
A.3	Kruskal-Wallis H test	186
A.4	Shapiro-Wilk test	187
A.5	Spearman's rho rank correlation test	188
B	Statistical Classification	189
B.1	Linear regression	189
B.2	Linear Support Vector Classifier	190
B.3	Logistic Regression	191
B.4	k-Nearest Neighbour Classification	192
B.5	k-fold Cross-validation	192
	Bibliography	195

List of Figures

1.1 Hypothesis driven knowledge discovery	2
1.2 Screenshot of a model written within Infobiotics Workbench (IBW2)	8
1.3 Screenshot of the model analysis results page from the ssapredict . .	9
2.1 Peptide synthesis	15
2.2 Partial propensity direct method data structures	32
2.3 Composition and rejection algorithm for random variate generation	35
2.4 Genetic device functioning as an AND gate	41
2.5 Genetic device functioning as an OR gate	42
2.6 GFP expression in the AND & OR gates over time	45
2.7 Heat map visualisations of the AND & OR gate transfer functions . .	46
2.8 Algorithm benchmark performance results in <i>rps</i> for AND gate	47
2.9 Algorithm benchmark performance results in <i>rps</i> for OR gate	48
3.1 Aggregation of <i>Dictyostelium discoideum</i>	53
3.2 cAMP production during <i>Dictyostelium</i> aggregation	54
3.3 Results and SSA benchmark for repeated cAMP oscillation experiment	55
3.4 Heat shock response in <i>Escherichia coli</i> model with SSA benchmark .	57
3.5 AI-2 pathways in <i>Escherichia coli</i> model with SSA benchmark	59
3.6 ABA signalling pathways in response to cellular dehydration	60
3.7 ABA regulation model with SSA benchmark	61
3.8 Auxin transport proteins in <i>Arabidopsis</i> cells	63
3.9 Auxin transport experiment model	63
3.10 SSA benchmark for the auxin transport model	64
3.11 Competing G protein-coupled receptor kinase signalling model . . .	65

3.12 SSA benchmark for the competing G protein-coupled receptor kinase model	67
3.13 Results of the Shnerb et al. discrete proliferation model	68
3.14 SSA benchmark of the Shnerb et al. discrete proliferation model . .	70
3.15 Model of lacZ lacY gene expression with SSA benchmark	71
4.1 Histogram of model reaction size within the BioModels dataset . . .	77
4.2 Example graph	79
4.3 Reaction Dependency Graph (RDG) generated from exemplar system	81
4.4 Species Network Graph (SNG) generated from exemplar system . . .	82
4.5 Graph with weakly connected components	85
4.6 Graph with articulation points and biconnected components	86
4.7 Graph possessing a high betweenness vertex	89
4.8 Directed cyclic graph with a girth of 3	90
4.9 Graph with a closed triple and an open triple	92
4.10 Heatmap of property value correlations for the BioModels dataset . .	94
4.11 Hierarchically clustered heatmap of property value correlations for the BioModels dataset	95
4.12 Property values and statistics of models in the BioModels and curated models datasets	100
4.13 Property values and statistics of models in the BioModels and curated models datasets	101
5.1 Overview of benchmarking suite	108
5.2 DSMTS birth-death process model “gold standard” results	112
5.3 DSMTS test for DM using model dsmts-001-01	113
5.4 DSMTS tests for FRM, LDM, SDM & ODM using model dsmts-001-01	115
5.5 DSMTS tests for NRM, PDM, CR & TL using model dsmts-001-01 . .	116
5.6 Histogram of winning algorithms for the BioModels dataset	117
5.7 Comparison of algorithm performance for every model in the BioModels dataset	118

5.8	Bi-clustered heatmap showing the performance of all 9 algorithms for every model in the BioModels dataset	121
6.1	Comparison of real and estimated performance for each algorithm and all BioModels using linear regression	133
6.2	Distribution of relative performance loss caused by mispredictions .	143
6.3	Algorithmic performance for an <i>Escherichia coli</i> quorum sensing circuit	145
7.1	Structural diagram of ssapredict architecture and work-flow	152
7.2	Screenshot of ssapredict “home” page	153
7.3	Cropped screenshot of ssapredict “home” page after model uploaded	154
7.4	Cropped screenshot of ssapredict results page	155
7.5	Cropped screenshot of ssapredict simulation settings page	156
7.6	Diagram of ssapredict file system source tree	159
7.7	Diagram of propertygen file system source tree	161
7.8	Function call graph of ssapredict python linear SVC prediction module	162
8.1	Terminal window showing example use of the ngss simulator	164
8.2	Terminal window containing a ngss XML simulation parameters file .	166
8.3	Diagram of ngss file system source tree	167
8.4	C++ class diagram of ngss SSA implementations	169
8.5	Diagram of ngss include directories on the file system	171
8.6	Code fragment from the ngss SimulateAlgorithmOpenMP function	175
8.7	Code fragment from the ngss SimulateAlgorithmMPI function . .	176
B.1	Optimal hyperplane and margins for SVM trained with 2 classes . . .	190

List of Tables

2.1	Propensity calculations for elementary reaction types	19
2.2	“Toy” reaction network	22
2.3	Propensity calculations	22
2.4	Kinetic rules for the Boolean AND gate	43
2.5	Kinetic rules for the Boolean OR gate	43
3.1	Summary of models available in the curated models dataset	52
4.1	Example reaction network and reaction dependencies	80
4.2	Summary of analysed model topological properties	84
5.1	Summary of available SSAs in benchmarking suite	109
5.2	Number of times one of the 4 best performing algorithms was ranked below the top 4 for the BioModels dataset	119
5.3	Distribution of best performing algorithms for the BioModels dataset	124
5.4	Algorithm ranking results from Mann Whitney U test using mean performance	125
5.5	Algorithm ranking results from Mann Whitney U test using perfor- mance values from all runs	125
6.1	Frequency of properties highlighted by feature selection methods . .	134
6.2	10-fold cross-validation experiment using feature selected properties	136
6.3	10-fold cross-validation experiment using intersection of fast and fea- ture selected property sets	137
6.4	10-fold cross-validation experiment using fast properties	138
6.5	10-fold cross-validation experiment using complete set of properties	138
6.6	Prediction experiment using feature selected properties	139

6.7	Prediction experiment using intersection of fast and feature selected property sets	140
6.8	Prediction experiment using fast properties	140
6.9	Prediction experiment using complete set of available properties . .	141
6.10	Network sizes for large scale model prediction experiments	145
7.1	Summary of fast model topological properties analysed by <i>propertygen</i>	157

For my parents, Vel and Mangkayarkarasi.

Chapter 1

Introduction

1.1 Background and motivation

Ever since the technology to sequence DNA became available, we entered the modern age of biological discovery and understanding. DNA, the codex of life, provides a rudimentary description of a biological organism. However, this description does not explicitly communicate the behavioural complexity of the biological interaction networks it encodes. A process such as morphogenesis [1] reveals the incredible capabilities of these interaction networks and the highly accurate timing and robustness required of these networks.

Systems biology [2] is a discipline that deciphers the internal mechanisms of complex biological systems using modelling and simulation techniques. Such biosystems are “black boxes” for scientists who may initially know a subset of the internal components but not how they interact to self-regulate. Models are formulated to represent hypotheses for how the system may behave, and are executed via simulation *in silico* for “dry” experimentation. Simulation results can be compared to real world “wet” experimental data to test hypothesis validity. Repeated hypothesis testing in this manner allows scientists to continuously refine their understanding and hone in on the complex biological reality (see Figure 1.1). Unfortunately, there is an inverse relationship between model complexity (i.e. the level of biological knowledge) and

simulation tractability. Therefore, simulation performance is an important limiting factor for knowledge acquisition in the field of systems biology.

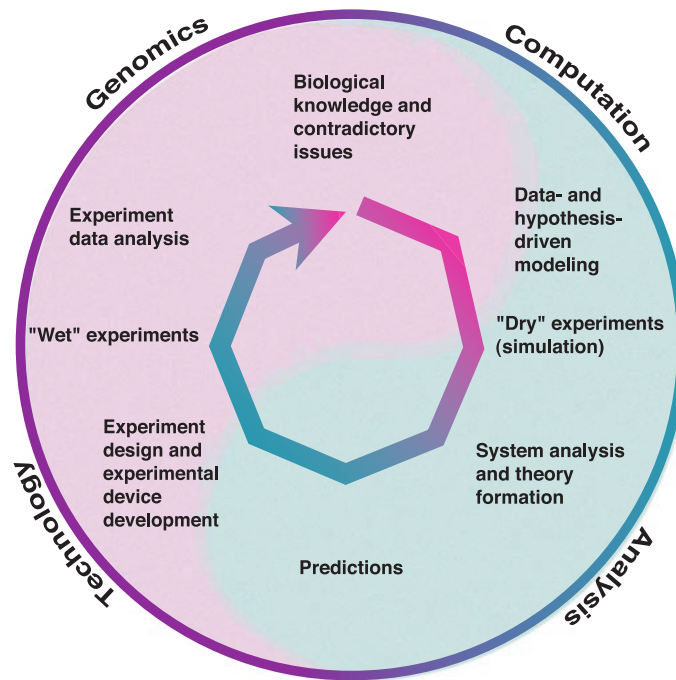


FIGURE 1.1: Hypothesis driven knowledge discovery (taken from [2])

Synthetic biology [3] embodies the idea of “*artificial life*”. This field takes two major research paths to reach this goal:

1. Synthetically created chemical systems that emulate the complex behaviour and properties of natural biochemical systems [4].
2. Designing and building complex living biosystems that would otherwise not exist in nature [5].

In this thesis I will discuss challenges related to the design of synthetic organisms. This area of synthetic biology considers cells as programmable information processing devices akin to computational devices. A major challenge is to engineer cells that perform useful behaviours which are not seen in naturally evolved organisms. Current synthetic biology builds novel biosystems using catalogued genetic components in a similar way to using Lego bricks. This approach has been typified by the BioBrick Foundation component approach and the annual iGEM competition that challenges budding synthetic biologists to create new biosystems based on BioBricks [6]. The

component abstraction simplifies the construction of more complex synthetic biosystems in the same way that a CPU designer no longer has to work at the silicon level, but instead works at higher levels of abstraction and connects components such as logic gates. This significantly reduces the potential design space and means that a synthetic biologist can create large or complex systems more quickly.

When genetic components are put together for a new synthetic biological design there may be unforeseen issues due to the underlying complexity of the biochemical interaction network. Furthermore, biological systems at the scale of a cell are prone to stochastic noise and the robustness of design needs to be evaluated. Therefore, it is necessary to generate models of the synthetic biological system and test the designs using simulation. Once the design has been refined from *in silico* modelling and simulation, a wet lab implementation can be created. Wet lab work is costly in terms of both finance and person-hours, therefore *in silico* knowledge can save large amounts of wet lab trial and error.

Biological systems are commonly modelled with ordinary differential equations (ODE), which is a continuous deterministic approach. ODEs can accurately model test tube scale chemical interactions, but are inaccurate for chemical systems with low molecular populations such as cells [7–9]. Stochastic Simulation Algorithms (SSAs) are the primary means of simulating naturally discrete cellular systems affected by stochastic noise, generating multiple *realistic* trajectories of molecular quantities over time from a set of reactions (with associated stochastic rate constants), initial amounts and stopping criteria. *Exact* SSAs, introduced by Gillespie [10], generate trajectories that are demonstrably equivalent to the *Chemical Master Equation* and must simulate each and every reaction in the system. The algorithmic complexity scaling of $O(M)$ (where M is the set of reactions), and concomitant generation of pseudo-random numbers to emulate stochasticity for each reaction event, makes simulating ever larger reaction networks increasingly intractable despite continued advances in computational power. Subsequently *approximate* SSAs have been introduced that conditionally apply multiples of reactions at each step [11].

My research group colleagues have been modelling and analysing biological systems [12, 13] whilst also investigating the challenges of designing synthetic life [14, 15] and other biochemical systems [16, 17]. This includes researching the auto-generation of synthetic bio-systems [18] using databases of modular genetic “components” [19] to generate complex systems with defined behaviour [20]. There has been a particular focus on developing tools that aid the design of synthetic bio-systems [21]. These tools perform hypothesis testing via model execution which involves the research, development and use of the SSA [22–24].

1.2 Aims and scope

The goal of this thesis is to aid scientists in the fields of systems and synthetic biology to use SSAs when simulating models of their biosystems. ODE models are the standard paradigm when modellers approach biosystems, but in many cases they are not suitable for such models. Continuous, deterministic ODE models are appropriate for test tube sized systems [7–9], but *in vivo* biosystems are actually composed of small compartmentalised units: **cells**. Cells are subject to stochastic noise because of the relatively small molecular population of each cell. Furthermore, by correctly considering discrete (molecular) entities and being provably equivalent to the CME, the SSA can be considered an exact trajectory of a biosystem rather than an approximation.

However, there are currently many compromises or drawbacks that have to be considered when using SSAs. Perhaps the most important of these reasons is that computational performance of the SSA may dissuade a scientist from using this technique. For example, if one considers a tissue scale system of multiple adjacent cells or a bacterial colony model involving perhaps thousands of cells, the large yet intricate reaction network may simply become intractable to compute. One of the strengths of the SSA is that it considers each and every reaction that occurs in the system as a discrete event. Unfortunately, this feature becomes a critical bottleneck when faced with systems that involve high molecular populations of reactive species.

A number of research groups have been working on improving the performance of the SSA with algorithmic improvements and strategies to reduce computational complexity. Consequently, many different variants of the SSA [11, 25–30] have been produced since the original Gillespie direct method [31] that claim to ameliorate computational performance. From a survey of the literature, I realised that many published SSAs are tested with an insufficient number of models, mostly tailored to properties of the newly introduced algorithm. Therefore, it is hard to extrapolate or compare performance between algorithms as each will often be benchmarked against competitors' algorithms using only favourable models. I have found that SSAs which claim to be “state of the art” may perform worse than supposedly less advanced variants with certain types of model. This notion is introduced as the first of three hypotheses evaluated in this thesis:

Hypothesis 1

There is no single SSA that is superior in performance for every biomodel

The cost of simulating a system with one SSA variant or another depends on the properties of the underlying network and the states reached during the simulation. Each biological model exhibits characteristics that may be suited to a particular simulation algorithm, such as the *degree of coupling*¹ in the reaction network or whether the system is especially *stiff*². Effective discrimination between SSAs should be based on matching favourable algorithmic properties to model characteristics. This leads us to the second hypothesis that shall be evaluated in this thesis:

Hypothesis 2

There is a relationship between biomodel characteristics and SSA performance

Through experimentation, I have found that SSA execution times can vary by several orders of magnitude depending on the model simulated. I have also noted a tendency for scientists to select one particular algorithm and use that for their simulations.

¹Degree of coupling is the maximum number of reactions in a reaction network that are affected by a reaction firing [29]

²Stiffness is caused by multiple processes occurring at differing time-scales within a reaction network [32].

Often this selection is based on some notion of intuition, algorithm availability from a software package, or even ease of algorithm implementation. Such an arbitrary selection of SSA could, in a bad case, result in a simulation taking in the order of months rather than hours to complete. If *hypothesis A* holds, I will be unable to find a single fastest algorithm to recommend for all models and therefore one must deduce the fastest SSA on a per model basis. If *hypothesis B* also holds, I should be able to quantify the best algorithm for a given model. Thus, this generates the final hypothesis that is evaluated in this thesis:

Hypothesis 3

An algorithm can select the best SSA for an arbitrary model with only a small margin of error

1.3 Structure of thesis

Chapter 2 outlines the background for the stochastic simulation algorithm.

Chapter 3 introduces modelling and simulation for systems and synthetic biology.

Chapter 4 presents a performance benchmark of 9 major stochastic simulation algorithm formulations over a large number of biomodels.

Chapter 5 presents an analysis of biochemical model characteristics.

Chapter 6 demonstrates the automated selection of the highest performing stochastic simulation algorithm for a given model.

Chapter 7 presents the stochastic simulator developed during the period of research. Next generation simulator (ngss) uses the implementations of the 9 algorithms benchmarked.

Chapter 8 presents the ssapredict meta simulator web application.

Chapter 9 concludes the thesis.

1.4 Main contributions

The work presented in this thesis contributes to two EPSRC funded synthetic biology projects: “ROADBLOCK: Towards Programmable Defensive Bacterial Coatings & Skins” (EP/I031642/1) and “AUDACIOUS: Towards a Universal Biological-Cell Operating System” (EP/J004111/1). Three pieces of software have been developed as part of this research work: (1) *ngss* (2) *ssapredict* (3) *SSA benchmarking suite*.

1.4.1 *ngss*: Next Generation Stochastic Simulator

ngss is the stochastic simulator I have developed as a major deliverable for this thesis. The simulator is an important component of the latest incarnation of the *Infobiotics Workbench (IBW2)* which is being developed for the *EPSRC ROADBLOCK* synthetic biology grant (see Figure 1.2). *ngss* allows for model designs written in the *Infobiotics Language (IBL)* to be executed and thus hypothesis tested prior to biomatter compilation. *ngss* currently implements nine different variants of the SSA: Direct Method (DM) [31], First Reaction Method (FRM) [10], Next Reaction Method (NRM) [25], Optimised Direct Method (ODM) [26], Sorting Direct Method (SDM) [27], Logarithmic Direct Method (LDM) [28], Partial Propensities Direct Method (PDM) [29], Composition Rejection (CR) [30] and Tau Leaping (TL) [11]. Model files can be loaded in the community standard *SBML* (systems biology modelling language) [33] format, or in the IBW2 “data-model” XML format. The simulator is able to output timeseries data in the ubiquitous CSV (comma separated values) format for simple import into analytical software. Compressed *HDF5* (hierarchical data format) [34] output is available for heavy duty and high performance computing applications.

The software is written in the C++ programming language with an emphasis on computational performance. For multi-core machine parallelism, *ngss* supports *OpenMP* [35] to distribute individual simulation runs on separate CPU cores. For computing cluster applications, *ngss* also supports *OpenMPI* [36]. Object oriented design principles were strictly adopted to ease the addition of new algorithms to the software. *ngss*

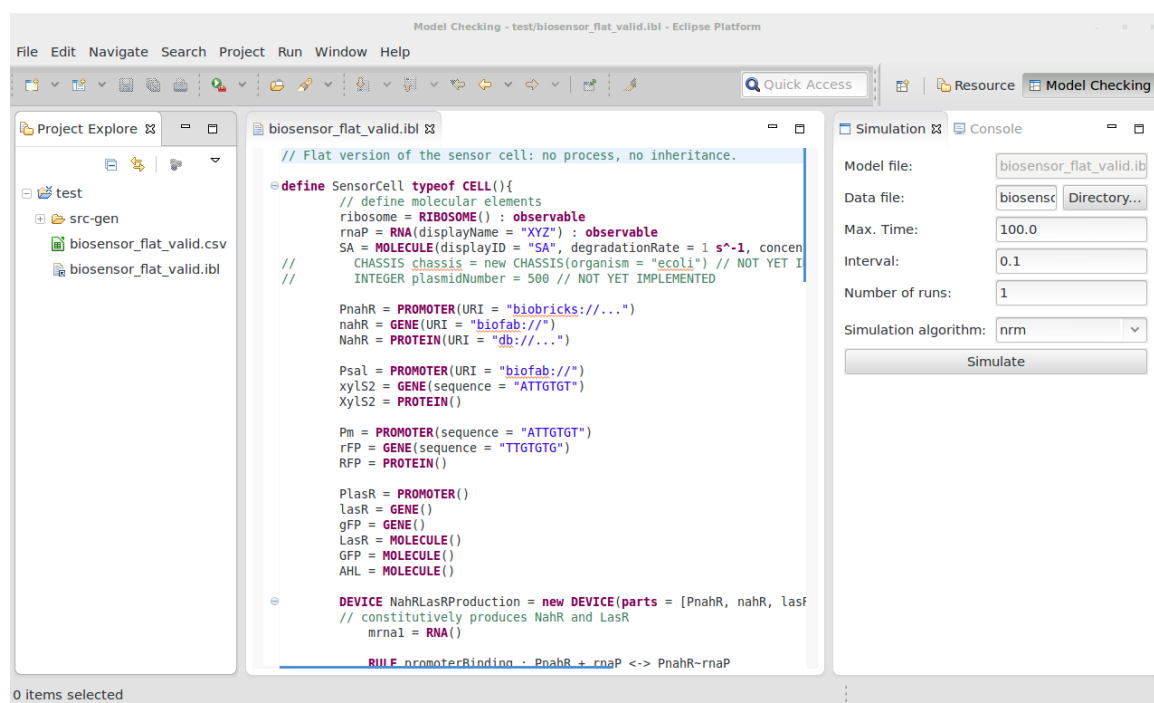


FIGURE 1.2: Screenshot of a model written in IBL within Infobiotics Workbench (IBW2) ready to be simulated with ngss. The view on the right hand side of the application is the *simulation* pane, where simulation parameters (including algorithm selection) for the model can be supplied before execution with ngss.

is open source software released under the terms of the GNU General Public License (GPL) version 3 and is available for Windows, Mac and Linux operating systems. The simulator source can be downloaded from <http://ssapredict.ico2s.org/resources>.

1.4.2 ssapredict

ssapredict is a web service designed to automate the process of determining the fastest SSA for a given model. This tool was designed to improve the usability and availability of SSAs for scientists. For example, uploading a model in SBML format is a one click operation. After the upload is complete, *ssapredict* uses trained classifiers to predict the fastest performing SSA based on the topological properties of the model. Once a prediction has been received, the scientist has the option to simulate the model (see Figure 1.3). With minimal effort the user can receive a statically built version of the ngss simulator for their operating system preconfigured to run their model with the optimal SSA selection.

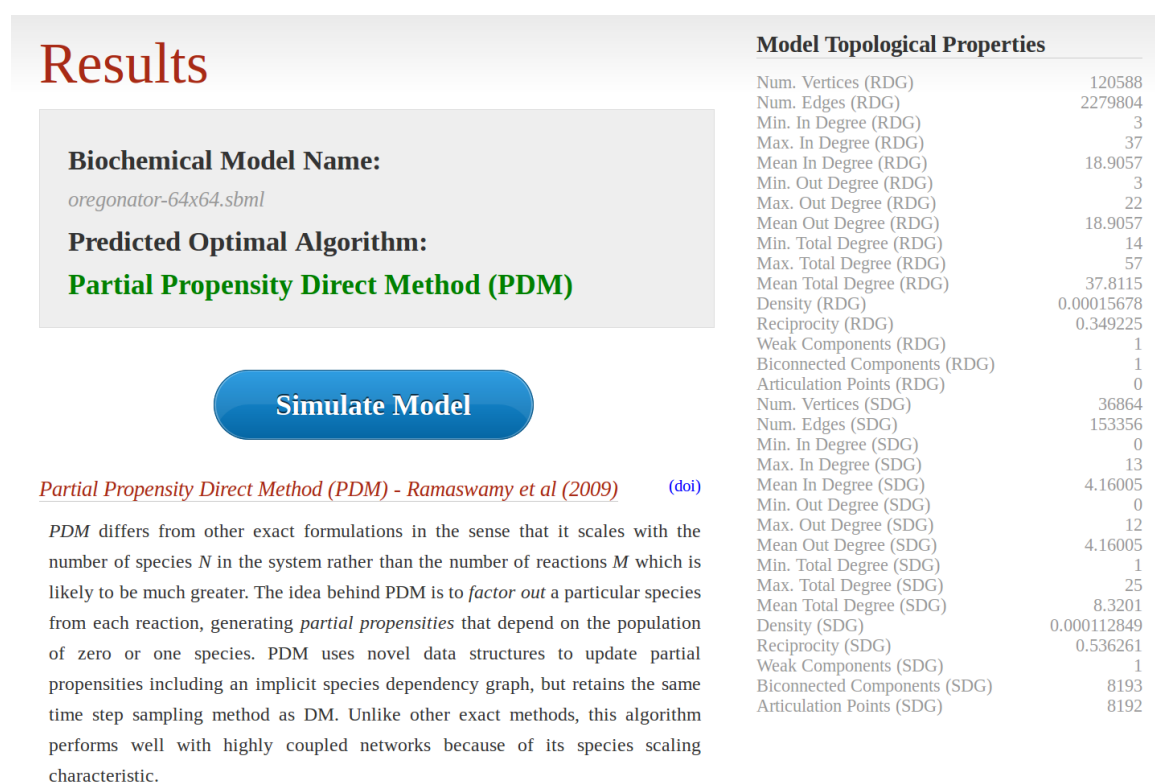


FIGURE 1.3: Screenshot of the model analysis results page from the ssapredict web application. Model topological property values are shown as well as the information about the predicted fastest SSA for this model. There is an option to simulate the model using ngss.

ssapredict is open source software released under the terms of the GNU Affero General Public License (AGPL). It is written using the python programming language using the web2py web application framework. The source code can be downloaded from <http://ssapredict.ico2s.org/resources>.

1.4.3 SSA benchmarking suite

The benchmarking suite is designed to be a tool for researchers that either use SSAs to simulate biochemical reaction networks or create new variants of the SSA. Scientists can evaluate their SBML format models using the suite's model metric analytics and assess which of the nine implemented algorithms is most suitable for their model. Software developers are able to implement their own algorithms and test them in the suite against other implemented algorithms for correctness, performance and memory usage. Furthermore, developers can use the source code for the supplied

algorithms in their own software. The benchmarking suite is released under the terms of GNU General Public License (GPL) version 3.

1.5 Published work

Journal articles

- Daven Sanassy, Paweł Widera, and Natalio Krasnogor. “Meta-Stochastic Simulation of Biochemical Models for Systems and Synthetic Biology”. *ACS Synthetic Biology*, 4(1):39–47, 2015.
- Savas Konur, Marian Gheorghe, Harold Fellermann, Daven Sanassy, Natalio Krasnogor, Christophe Ladroue, Sara Kalvala, Laurentiu Mierla, Florentin Ipate. “In Silico Design and Analysis of Genetic Boolean Gates: Membrane Computing Approach”, *J. Theor. Comp. Sci.* (submitted).

Conference papers

- Daven Sanassy, Harold Fellermann, Natalio Krasnogor, Savas Konur, Laurentiu M. Mierla, Marian Gheorghe, Christophe Ladroue, Sara Kalvala. “Modelling and Stochastic Simulation of Synthetic Biological Boolean Gates”. *Proceedings of 16th IEEE International Conference on High Performance Computing and Communications*, pp. 404-408, Paris, France, 2014.
- Savas Konur, Christophe Ladroue, Harold Fellermann, Daven Sanassy, Laurentiu Mierla, Florentin Ipate, Sara Kalvala, Marian Gheorghe, Natalio Krasnogor. “Modeling and Analysis of Genetic Boolean Gates using Infobiotics Workbench”. *Proceedings of Workshop on Verification of Engineered Molecular Devices and Programs 2014 (CAV 2014)*, pp. 26-37, Vienna, Austria, 2014.

Extended abstracts

- Daven Sanassy, Jonathan Blakes, Jamie Twycross, Natalio Krasnogor. “Improving Computational Efficiency in Stochastic Simulation Algorithms for Systems and Synthetic Biology”, Proceedings of SynBioCCC: Workshop on Design, Construction, Simulation and Testing of Synthetic Gene Regulatory Networks for Computation, Control, and Communications, pp. 1-4, Paris, France, 2011.

Chapter 2

Background Theory

2.1 Introduction

Simulation of mathematical and computational models of reaction networks is an invaluable tool for biologists aiming to understand the dynamic behaviour of complex biochemical systems. In the fields of Systems and Synthetic Biology, repeated rounds of model-driven hypothesis generation, validated or refuted by wet lab experimentation, lead to refined quantitative and predictive models. *In silico* experimentation with these models is cheaper, faster, and more reproducible than its physical counterpart.

2.2 Biological overview

Biological systems are biochemical “machines” that exist for the primary function of replication of the instruction set that encodes them [37]. “Replication machines” are typically realised as (one or many) biological cells which provide a closed structural environment to enable replication behaviour and survival. These replication machines also each encapsulate a copy of the very “code” that describes them; this

code is stored as the famous DNA (deoxyribonucleic acid) molecule. There are subsections of DNA that encode for protein molecules called *genes*. Proteins have many biochemical functions and are the method of control that genes possess in order to modulate the functions of the cell.

Biological cells can be considered as information processing devices [38], comparable to a programmable electronic appliance [39]. Whilst electronic devices are made up of circuits that regulate switching electron flow to produce complex behaviour, biological cells are made up of *genetic circuits* whose communication currency is molecules (rather than electrons). Genetic circuits are subcomponents of the overall gene regulatory network of an organism. Gene regulatory networks describe the interactions of the organism's molecular species and dictate behaviour via modulation of gene products. Gene products are the result of *gene expression* which is a discrete stochastic process. This process begins with a subsection of the DNA sequence called a "*transcription unit*" which encapsulates the transcription of gene(s) to RNA (Ribonucleic acid). The transcription unit has a "*promoter*" site at the beginning of its sequence, followed by the region to transcribe and a stopping sequence. The promoter allows for the initialisation of gene expression by providing a site for the enzyme *RNA polymerase* (RNAP) to bind. RNAP "reads" through the coding region of the DNA and rewrites this code into different forms of RNA. One form of RNA generated is messenger RNA (mRNA) which provides the instructions for protein synthesis. Transcription is regulated by proteins called *transcription factors* which control the rate of RNA production. Transcription factors upregulate or downregulate RNA production by altering the binding affinity of RNAP with the promoter region of the transcription unit. There are two classes of transcription factor: (1) *repressors* which downregulate and (2) *activators* which upregulate.

A piece of molecular machinery called a *ribosome* subsequently translates mRNA into proteins (see Figure 2.1). Translation is initiated after the ribosome attaches to the ribosome binding site (RBS). The RBS is a section of the mRNA that is responsible for binding to the ribosome. The ribosome reads through the mRNA sequence which encodes the sequence of amino acids that form a polypeptide chain (i.e. a protein).

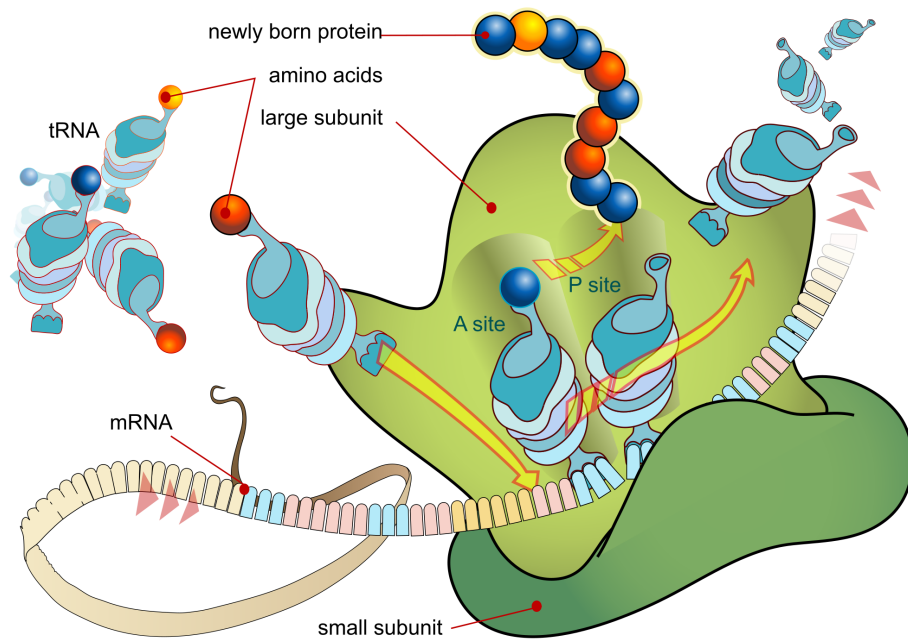
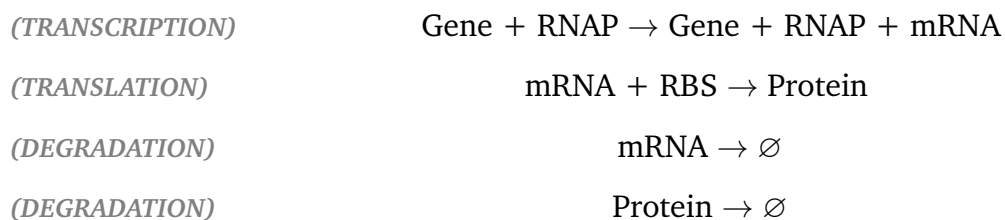


FIGURE 2.1: Peptide synthesis - the (green) ribosome reads through the mRNA which is translated to a peptide chain (taken from [40]). tRNAs recruit amino acids to the ribosome which are assembled in the order specified by the mRNA template to the growing peptide chain.

Every three codons (letters) of the mRNA sequence encodes an amino acid, which the ribosome attaches to the developing polypeptide chain using transfer RNA (tRNA). When this translation process is terminated by the stopping sequence, the polypeptide chain will then fold into a very precise protein structure. Bio-components of the system can be *degraded*, for example mRNA degradation is catalysed by the *Ribonuclease* (RNase) enzyme. Degradation is an important system process to maintain homeostasis.

System behaviour can be *modelled* by creating reaction *rules* of a biochemical process. As an illustration, the gene expression process can be modelled by the following rules:



The left-hand-side of a rule states the *reactant* molecular species, whilst the right-hand-side describes the *product* species. Note that the level of system detail is chosen by the modeller, and may be limited by biological knowledge. To simplify the model, one may combine several processes into a single rule. For example, the current gene expression rules implicitly include the tRNA mechanisms. The tRNA mechanism could be explicitly added to the model as extra rules if finer detail is required.

2.3 Biochemical modelling

Biochemical systems are traditionally described by mathematical models in the form of Ordinary Differential Equations (ODEs), namely Reaction Rate Equations (RREs) [31]. These ODEs assume the system to be deterministic and the system's molecular populations to be continuous variables. This is in stark contrast to reality where molecules are discrete entities best represented as integer values, and where the chemical kinetics of the system are non-deterministic. Whilst it may not seem intuitive to model in this manner, RREs can give a remarkably accurate result for the temporal evolution of the system when molecular populations are large [7]. However, there are several reasons that mean the RREs may be inappropriate for modelling the full range of biochemical systems. When the populations of chemical species are low, this deterministic approach cannot properly account for the stochastic noise present in the system that is inversely proportional to the square root of the size of the molecular population [31]:

$$\text{noise} \sim \frac{1}{\sqrt{\#molecules}} \quad (2.1)$$

Another issue with RREs is treating the molecular populations as continuous when in reality these are discrete. For example, a discrete model would allow a gene to be in either an active or inactive state, whilst in a continuous model a gene is erroneously considered to always be fractionally active. Switching between active and inactive states as regulators bind and unbind, creates bursts of transcription,

the timing and frequency of which is unpredictable. Understanding such events is critically important to the emerging field of synthetic biology where the emphasis is on control and reliability of genetically encoded systems. The inability of RREs to capture fluctuations introduced by stochastic noise and thus potentially different trajectories of the system are a major shortcoming. Furthermore, for complex systems RREs are not guaranteed to give an accurate average of the molecular populations [31].

The aforementioned issues are strong evidence that it is desirable to model the dynamics of a biochemical system as the discrete, stochastic process that it is in reality. A chemical reaction can only occur if the reactant molecules moving with Brownian motion collide in the correct orientation and with sufficient energy. Therefore, one can model a biochemical system by the probabilities of specific types of molecular reactions occurring within a time interval. In fact, Gillespie defines the fundamental hypothesis of *stochastic chemical kinetics* as the “average probability that a particular combination of reactant molecules will react accordingly in the next infinitesimal time interval dt ” [31]. It is important to note that this relies on the assumption that the system is both *well-stirred* and thermally equilibrated in order to simply calculate a reaction probability from just a stochastic reaction constant and knowledge of the molecular populations.

There are two different approaches to stochastically modelling biochemical systems. The first is the stochastic Chemical Master Equation (CME) which is typically an infinite set of ODEs derived from the fundamental hypothesis [27]. Solving the CME requires the consideration of every single possible simulation trajectory and cannot be solved analytically or numerically for all but the most trivial cases. However, one can “kinetically sample” [41] the CME using the Stochastic Simulation Algorithm (SSA), which is mathematically equivalent to the CME but derived independently from the fundamental hypothesis. The SSA is a computational method that models molecules as discrete entities and their interactions as steps in an algorithm that can be executed (as opposed to solved mathematically) to simulate the system’s behaviour. This technique falls under the category of *executable biology* [42] or

algorithmic systems biology [43] and corresponds more closely to the way biologists think about molecules interacting. Simulating the mechanism by which the temporal evolution of the system occurs in a stepwise fashion allows one to understand how and why a system moves from one state to another [43]. This exact *mechanistic* reaction-by-reaction execution of the biosystem provides insights not available with the RREs (which can only approximate this behaviour) [31].

The second stochastic approach is to model the system using stochastic ordinary differential equations (SDEs). However, the SDE approach is continuous, and does not consider molecules as discrete entities. Therefore, this approach cannot provide the mechanistic account of a biosystem afforded by the SSA. Furthermore, SDE theory is “daunting for a typical applied mathematics student” [44] – a steep barrier of entry for a typical biologist.

2.4 Stochastic Simulation Algorithms (SSA)

2.4.1 Introduction

Gillespie first introduced the Stochastic Simulation Algorithm (SSA) as a novel Markov chain Monte Carlo simulation technique for chemically reacting systems in 1976 [10, 31]. He initially produced two formulations, the First Reaction Method (FRM) and the Direct Method (DM). Whilst both these algorithms are quite different in implementation, they are equivalent and share a common structure (see Algorithm 1).

Algorithm 1 Common algorithmic steps for SSAs

```
1: procedure SSA(molecular species, reactions)
2:   while reaction available to fire do
3:     calculate reaction propensities                                ▷ Step (1)
4:     select reaction to execute                                    ▷ Step (2)
5:     calculate reaction time                                       ▷ Step (3)
6:   end while                                                    ▷ Simulation time exceeded
7: end procedure
```

The SSA has 3 major steps per algorithmic iteration. Firstly, propensity calculations (Table 2.1) are required for every *reaction channel*¹ in the system. This is achieved by iterating through the reactions, and considering the reaction type, the stochastic rate constants and the number of reactants present in the system at that moment. The more reactants available in the system for a particular reaction, the greater its propensity (i.e. the probability of that reaction occurring).

Reaction type	Example	#Reactants	Propensity function
Source	$\emptyset \rightarrow A$	0	$a_j(x) = c_j$
Uni-molecular	$A \rightarrow B$	1	$a_j(x) = c_j x_1$
Bimolecular Homogeneous	$A + A \rightarrow B$	2	$a_j(x) = \frac{c_j x_1(x_1-1)}{2}$
Bimolecular Heterogeneous	$A + B \rightarrow C$	2	$a_j(x) = c_j x_1 x_2$

TABLE 2.1: Propensity calculations for elementary reaction types where $a_j(x)$ is the propensity of reaction j and c_j is the stochastic rate constant of reaction j . The variables x_1 and x_2 represent the species (amounts) involved in the reactions.

The last 2 algorithmic stages of the SSA rely on random sampling to select a reaction to fire and time interval to increment the simulation. As the FRM and DM diverge in how reactions are selected for execution and how the time intervals are generated, I shall present both algorithms in the following section to highlight their differences.

2.4.2 First Reaction Method & Direct Method

At each iteration, the FRM (Algorithm 2) calculates a time interval τ for each and every reaction in the system to fire, and chooses the reaction with the shortest τ to fire next. It should be noted that at each iteration, the τ of every reaction needs to be recalculated. The formula for calculating the τ for each reaction is shown in Equation 2.2:

$$\tau_j = \frac{1}{a_j(x)} \ln \left(\frac{1}{r_j} \right) \quad (2.2)$$

¹A reaction channel is a single stochastic model *rule* which implements reaction behaviour when executed.

Algorithm 2 First Reaction Method (FRM) [10]

```

1: procedure FRM(molecular species, reactions)
2:   set simulation time  $t \leftarrow 0.0$  ▷ initialise
3:   initialise species state vector  $X$  [1.. $N$ ]
4:   store vector of reactions  $R$  [1.. $M$ ]
5:   while  $t < t_{end}$  do
6:     for  $j \leftarrow 1$  to  $M$  do ▷ calculate reaction propensities
7:       calculate propensity  $a_j$ 
8:     end for
9:     for  $j \leftarrow 1$  to  $M$  do ▷ calculate a time for each reaction
10:      generate  $r_1 \leftarrow rand()$ 
11:       $\tau_j \leftarrow -1.0 * \ln(r_1)/a_j$ 
12:    end for
13:     $\mu \leftarrow j$  where  $j$  is  $\min\{\tau_j\}$  ▷ select reaction  $\mu$  to fire
14:    update state vector  $X \leftarrow X + R_\mu$  ▷ execute reaction
15:    update simulation time  $t \leftarrow t + \tau_\mu$ 
16:  end while
17: end procedure

```

A uniform random number r is required to calculate τ for each reaction j . The τ for a reaction j is related to the propensity $a_j(x)$ of it occurring. A reaction with a higher propensity is more likely to have a shorter τ .

The DM (Algorithm 3) takes an alternative, equally valid approach and calculates a single τ per iteration for the *next* reaction to occur by calculating the τ (see Equation 2.3) based on the total propensity, a_0 , of the system (see Equation 2.4).

$$\tau = \frac{1}{a_0(x)} \ln \left(\frac{1}{r_2} \right) \quad (2.3)$$

$$a_0(x) = \sum_{j=1}^M a_j(x) \quad (2.4)$$

This approach provides a drastic performance improvement on FRM, as only one uniform random number r_2 is required for τ calculation per iteration. Random number generation is typically computationally expensive [25] and should be minimised wherever possible. Whereas FRM scaled with $O(M)$ for random number usage, DM is $O(1)$ [27].

Algorithm 3 Direct Method (DM) [31]

```

1: procedure DM(molecular species, reactions)
2:   set simulation time  $t \leftarrow 0.0$  ▷ initialise
3:   initialise species state vector  $X$  [1.. $N$ ]
4:   store vector of reactions  $R$  [1.. $M$ ]
5:   while  $t < t_{end}$  do
6:     set total propensity  $a_0 \leftarrow 0.0$  ▷ calculate reaction propensities
7:     for  $j \leftarrow 1$  to  $M$  do
8:       calculate propensity  $a_j$ 
9:        $a_0 \leftarrow a_0 + a_j$ 
10:    end for
11:    generate  $r_1 \leftarrow rand()$  ▷ select reaction  $\mu$  to fire
12:    target propensity  $a_t \leftarrow a_0 r_1$ 
13:    for  $j \leftarrow 1$  to  $M$  do
14:       $a_t \leftarrow a_t - a_j$ 
15:      if  $a_t \leq 0$  then
16:         $\mu \leftarrow j$ 
17:        break
18:      end if
19:    end for
20:    update state vector  $X \leftarrow X + R_\mu$  ▷ execute reaction
21:    generate  $r_2 \leftarrow rand()$  ▷ calculate reaction time
22:     $\tau \leftarrow -1.0 * \ln(r_2) / a_0$ 
23:    update simulation time  $t \leftarrow t + \tau$ 
24:  end while
25: end procedure

```

Reactions are selected in Monte Carlo fashion where probability is proportional to propensity. One multiplies the total propensity a_0 of the system by a uniform random number r_1 , and performs linear search until the target value $r_1 a_0$ is reached (see Equation 2.5). The target propensity a_j found by the linear search dictates which reaction μ is selected.

$$\min \left\{ \mu \mid \sum_{j=1}^{j=\mu} a_j(x) > r_1 a_0(x) \right\} \quad (2.5)$$

Because DM is significantly more efficient than FRM, it was at that point in time considered the de-facto standard SSA formulation. However, the structure of FRM provides routes for SSA optimisation [25]. Gillespie highlighted the advantages of the algorithms: they are exact (ODEs can only approximate time increments between reactions) and accurately account for the noise present in the system. Moreover, the algorithms are simple to implement and have low memory requirements. However,

he also realised the disadvantages of the algorithms, they are computationally expensive for just a single run and a large number of runs are often required in order to have confidence in the averaged result.

NOTE: Uniform random numbers used by in reaction selection and τ calculation are in the interval $[0,1]$.

2.4.3 Worked through example of Direct Method for a simplified network

I shall now work through a toy example (see Table 2.2) for Direct Method to clarify the algorithm's execution. In the system, there are 3 reactions with 2 species with all parameters for the simulation listed.

Reaction	Rate Constant	Type
$\mathbf{G} \rightarrow \mathbf{P}$	1.0	Uni-molecular
$\mathbf{P} \rightarrow \emptyset$	0.1	Uni-molecular
$\mathbf{P} + \mathbf{P} \rightarrow \mathbf{P.P}$	0.7	Bimolecular Homogeneous

TABLE 2.2: "Toy" reaction network.

Let us set the initial amounts of $\mathbf{G} = 1$, $\mathbf{P} = 3$ and $\mathbf{P.P} = 0$ and the simulation time $t = 0.0$.

- The first step is to calculate the propensity for each reaction in the system.

Reaction	Propensity formula	Propensity calculation	Propensity $a_j(x)$
$\mathbf{G} \rightarrow \mathbf{P}$	$c_j x_1$	1.0×1	1.0
$\mathbf{P} \rightarrow \emptyset$	$c_j x_1$	0.1×3	0.3
$\mathbf{P} + \mathbf{P} \rightarrow \mathbf{P.P}$	$\frac{c_j x_1 (x_1 - 1)}{2}$	$\frac{0.7 \times 3 \times (3 - 1)}{2}$	2.1

TABLE 2.3: Propensity calculations

- The next step is to sum all the propensities in the system to get the total propensity $a_0(x)$ (see Equation 2.4), which is $a_0(x) = 1.0 + 0.3 + 2.1 = 3.4$.
- One now needs to generate a random number r_2 (using a uniform random number generator) in order to select a reaction (see Equation 2.5); I shall assume $r_2 = 0.5$. The total propensity is subsequently parametrised by the random number $r_2 a_0(x) = 0.5 \times 3.4 = 1.7$ for reaction selection.
- Now one must subtract reaction propensities $a_j(x)$ where $j[1..M]$, from the parametrised total propensity $r_2 a_0(x) = 1.7$ until $r_2 a_0(x) \leq 0.0$. The first reaction has propensity 1.0, resulting in $r_2 a_0(x) = 1.7 - 1.0 = 0.7$ and therefore not selected. Subtracting the second reaction propensity results in $r_2 a_0(x) = 0.7 - 0.3 = 0.4$, so is not selected. Subtracting the final reaction yields $r_2 a_0(x) = 0.4 - 2.1 = -1.7$ and is therefore selected.
- Following the selection of reaction 3, one applies it by changing the species amount vector. This reaction removes two of species **P** and adds one of species **P.P**. The state vector is therefore now $\mathbf{G} = \mathbf{1}$, $\mathbf{P} = \mathbf{1}$ and $\mathbf{P.P} = \mathbf{1}$
- The final step is to calculate the τ for this reaction application (see Equation 2.3). Another uniform random number r_1 must be generated (which one can assume to be 0.2 in this example) to be used for the τ calculation. $\tau = \frac{1}{3.4} \ln\left(\frac{1}{0.2}\right) = 0.473364092$. Simulation time is progressed $t = t + \tau$.
- This process is repeated until the maximum execution time is exceeded or there are no more reactions to fire (if total propensity $a_0(x) = 0.0$).

2.4.4 Next Reaction Method

The first major revision to the exact SSA, the Next Reaction Method (NRM, Algorithm 4) was published in 2000 by Gibson & Bruck [25] - more than 20 years after the original algorithms. Following renewed interest in stochastic modelling, the requirement to simulate ever larger networks increased. In spite of Moore's Law and the resulting improvements in computational power available, the existing algorithms did not scale satisfactorily to larger reaction networks and it was recognised that more efficient algorithms were key [25].

The NRM is based on FRM and introduces multiple algorithmic enhancements to greatly improve computational efficiency. A *dependency graph* for reactions is adopted (See Section 2.4.11), so that only affected reactions are considered when recalculating propensities. Thanks to the dependency graph, this step of the algorithm scales as $O(\log M)$ with loosely *coupled*² networks but still retains $O(M)$ worst case performance.

FRM's Achilles heel is its heavy use of random numbers (M per iteration), $M - 1$ of which are subsequently discarded. This is significant as Gibson & Bruck approximate generating one random to be roughly equivalent in computational expense to ten division operations [25]. NRM removes this wastage and instead employs an *indexed priority queue* [45] data structure to store unused τ values for use at the appropriate time. This is made possible by considering absolute, rather than relative, τ values for reaction execution. Gibson & Bruck demonstrated how absolute τ values allow legitimate reuse of statistically independent random numbers. Absolute τ values are simply calculated by generating a relative τ and adding it to the current simulation time. Because of this τ reuse, NRM only requires one random number to be generated per iteration, so this step of the algorithm has now been made $O(1)$ rather than $O(M)$ and is cheaper than DM which requires two random numbers per iteration. The authors claim that the indexed priority queue is a good choice in terms of

²Degree of coupling is the maximum number of reactions in a reaction network that are affected by a reaction firing [29]

Algorithm 4 Next Reaction Method (NRM) [25]

```

1: procedure NRM(molecular species, reactions)
2:   set simulation time  $t \leftarrow 0.0$  ▷ initialise
3:   initialise species state vector  $X$  [1.. $N$ ]
4:   store vector of reactions  $R$  [1.. $M$ ]
5:   initialise dependency graph  $DG$ 
6:   initialise indexed priority queue  $PQ$ 
7:   while  $t < t_{end}$  do
8:     if first iteration then ▷ calculate reaction propensities
9:       for  $j \leftarrow 1$  to  $M$  do
10:        calculate propensity  $a_j$ 
11:      end for
12:    else
13:      for  $dep$  in  $DG_\mu$  do
14:         $a_{old} \leftarrow a_{dep}$ 
15:        calculate updated propensity  $a_{dep}$ 
16:        if  $dep \neq \mu$  then
17:          if  $\tau_{dep} \neq \infty$  then
18:            if  $a_{dep} \neq 0.0$  then
19:               $R = a_{old} / a_{dep}$ 
20:               $\delta\tau = \tau_{dep} - t$ 
21:               $\tau_{new} = R \times \delta\tau + t$ 
22:            else
23:               $\tau_{new} = \infty$ 
24:            end if
25:          else
26:            if  $a_{dep} \neq 0.0$  then
27:              generate  $r_1 \leftarrow rand()$ 
28:               $\tau_{dep} \leftarrow -1.0 * \ln(r_1) / a_{dep}$ 
29:               $\tau_{new} = \tau_{dep} + t$ 
30:            end if
31:          end if
32:        end if
33:         $PQ_{dep} \leftarrow$  update with  $\tau_{new}$ 
34:      end for
35:    end if
36:    if first iteration then ▷ populate indexed priority queue
37:      for  $j \leftarrow 1$  to  $M$  do
38:        if  $a_j \neq 0.0$  then
39:          generate  $r_2 \leftarrow rand()$ 
40:           $\tau_j \leftarrow -1.0 * \ln(r_2) / a_j$ 
41:        else
42:           $\tau_j = \infty$ 
43:        end if
44:         $PQ_j \leftarrow$  store  $\tau_j$ 
45:        insert  $PQ_j$  node  $\rightarrow PQ$ 
46:      end for
47:    end if
48:     $PQ_{top} \leftarrow$  get top node from  $PQ$  ▷ select reaction  $\mu$  to fire
49:     $\mu \leftarrow$  get  $PQ_{top}$  reaction
50:    update state vector  $X \leftarrow X + R_\mu$  ▷ execute reaction
51:     $\tau \leftarrow$  get  $PQ_{top}$  absolute reaction time ▷ set absolute reaction time
52:    update simulation time  $t \leftarrow \tau$ 
53:  end while
54: end procedure

```

computational expense because of a low update count per iteration and as NRM only ever requires access to the highest priority element (reaction with lowest τ value).

Thanks to these changes, NRM scales with $O(\log M)$ for reaction selection, and can therefore handle much larger reaction networks than DM. It is important to remember that whilst NRM constitutes a massive leap in SSA computational efficiency, this comes at a cost. The new data structures (indexed priority queue and dependency graph) drastically increase memory requirements and implementing NRM correctly is significantly more challenging than deploying DM.

2.4.5 Optimised Direct Method

Cao et al. introduced the Optimised Direct Method (ODM, Algorithm 5) in 2004, as a modified version of DM claimed to outperform NRM. The idea behind the ODM is to sort reactions channels such that those with higher propensity values are assigned lower index values. Because larger reaction networks have a tendency to be *multi-scale* (they have some reactions that are far more likely to occur than others), sorting the reactions channels in this way reduces the *average search depth* of the DM reaction selection linear search. This is achieved by a short pre-simulation run of DM to assess the average propensity values of the reactions, after which it resumes the simulation with the reordered indexes [26].

The authors declared that for “real world” models they had tested, ODM outperformed NRM. However, they noted that they had not tested large enough models to fully realise the scaling advantage of NRM. They demonstrated that the degree of reaction network coupling severely affected NRM performance. A major complaint was that Gibson & Bruck had not specified how coupled a reaction network would have to be for NRM performance to begin deteriorating in comparison to DM. After profiling NRM with a favourable model for the algorithm, they discovered that the vast majority of computational time was spent maintaining the indexed priority queue (totally eclipsing the expense of random generation). This led them to realise that DM could be modified taking inspiration from NRM, without needing an

Algorithm 5 Optimised Direct Method (ODM) [26]

```

1: procedure ODM(molecular species, reactions)
2:   set simulation time  $t \leftarrow 0.0$  ▷ initialise
3:   initialise species state vector  $X$   $[1..N]$ 
4:   store vector of reactions  $R$   $[1..M]$ 
5:   initialise dependency graph  $DG$ 
6:   initialise reaction search order  $SO$   $[1..M] = 1..M$ 
7:   set total propensity  $a_0 \leftarrow 0.0$ 
8:   initialise pre-sim propensities  $RP$   $[1..M] = \{0\}$  ▷ pre-simulation
9:   while  $t < t_{presim}$  do
10:    run direct method for a timestep  $\tau$ 
11:    for  $j \leftarrow 1$  to  $M$  do
12:       $RP_j += propensity_j$ 
13:    end for
14:    update simulation time  $t \leftarrow t + \tau$ 
15:  end while
16:  sort  $RP$  to from highest to lowest ▷ set reaction search order  $SO$ 
17:  set  $SO$  to have  $RP$  reaction order
18:  set simulation time  $t \leftarrow 0.0$ 
19:  while  $t < t_{end}$  do
20:     $CalculatePropensities()$  ▷  $CalculatePropensities()$  described in Algorithm 6
21:    generate  $r_1 \leftarrow rand()$  ▷ select reaction  $\mu$  to fire
22:    target propensity  $a_t \leftarrow a_0 r_1$ 
23:    for  $j \leftarrow 1$  to  $M$  do
24:       $k \leftarrow SO_j$ 
25:       $a_t \leftarrow a_t - a_k$ 
26:      if  $a_t \leq 0$  then
27:         $\mu \leftarrow k$ 
28:        break
29:      end if
30:    end for
31:    update state vector  $X \leftarrow X + R_\mu$  ▷ execute reaction
32:    generate  $r_2 \leftarrow rand()$  ▷ calculate reaction time
33:     $\tau \leftarrow -1.0 * \ln(r_2) / a_0$ 
34:    update simulation time  $t \leftarrow t + \tau$ 
35:  end while
36: end procedure

```

indexed priority queue. Firstly, the dependency graph from NRM was adopted as this does not have an associated update cost per iteration whilst improving propensity update efficiency. Secondly, the aforementioned modifications to reaction channel sorting are implemented in order to vastly improve reaction search performance in “undoubtedly” multi-scale real world reaction networks [26].

Algorithm 6 CalculatePropensities()

```

1: function CALCULATEPROPENSITIES
2:   if first iteration then
3:     for  $j \leftarrow 1$  to  $M$  do
4:       calculate propensity  $a_j$ 
5:        $a_0 += a_j$ 
6:     end for
7:   else
8:     for  $dep$  in  $DG_\mu$  do
9:        $a_0 -= a_{dep}$ 
10:      calculate updated propensity  $a_{dep}$ 
11:       $a_0 += a_{dep}$ 
12:    end for
13:   end if
14: end function

```

2.4.6 Sorting Direct Method

The Sorting Direct Method (SDM, Algorithm 7) was introduced by McCollum et al in 2006 [27] as a natural successor to ODM. They note that whilst ODM appeared to be the fastest SSA variant, it suffered from the inability to deal with sharp transient changes in reaction propensities that can occur in biological systems. The change proposed abandoning the pre-simulation aspect of ODM and instead opted for an efficient dynamic analysis of reaction propensities by allowing the system to be loosely sorted at runtime. The authors claim that SDM always performs at least as well, if not better than ODM when benchmarked against real world models [27].

McCollum et al. demonstrate that the assumption made by the ODM, that long term reaction execution behaviour will not change, is incorrect, for example in an oscillating system or one that is affected by a burst of transcriptional activity. It is likely that the authors of ODM realised this, but were concerned about the potential cost of continuously sorting reaction channels. SDM achieves high performance by only approximately sorting the reaction channel indexes. Instead of a full sort per iteration, the reaction which fired is moved up in reaction order to the next lowest index. This only requires a pointer swap of two memory addresses per iteration, and this loose style of sorting adds very little computational expense to the algorithm. Testing performed by the authors of SDM demonstrate a performance advantage for

Algorithm 7 Sorting Direct Method (SDM) [27]

```

1: procedure SDM(molecular species, reactions)
2:   set simulation time  $t \leftarrow 0.0$  ▷ initialise
3:   initialise species state vector  $X$  [1.. $N$ ]
4:   store vector of reactions  $R$  [1.. $M$ ]
5:   initialise dependency graph  $DG$ 
6:   initialise reaction search order  $SO$  [1.. $M$ ] = 1.. $M$ 
7:   set total propensity  $a_0 \leftarrow 0.0$ 
8:   while  $t < t_{end}$  do
9:      $CalculatePropensities()$  ▷  $CalculatePropensities()$  described in Algorithm 6
10:    generate  $r_1 \leftarrow rand()$  ▷ select reaction  $\mu$  to fire
11:    target propensity  $a_t \leftarrow a_0 r_1$ 
12:    for  $j \leftarrow 1$  to  $M$  do
13:       $k \leftarrow j$ 
14:       $l \leftarrow SO_j$ 
15:       $a_t \leftarrow a_t - a_l$ 
16:      if  $a_t \leq 0$  then
17:         $\mu \leftarrow l$ 
18:        break
19:      end if
20:    end for
21:    if  $k \neq 0$  then ▷ update reaction search order  $SO$ 
22:       $tmp \leftarrow SO_k$ 
23:       $SO_k \leftarrow SO_{k-1}$ 
24:       $SO_{k-1} \leftarrow tmp$ 
25:    end if
26:    update state vector  $X \leftarrow X + R_\mu$  ▷ execute reaction
27:    generate  $r_2 \leftarrow rand()$  ▷ calculate reaction time
28:     $\tau \leftarrow -1.0 * \ln(r_2) / a_0$ 
29:    update simulation time  $t \leftarrow t + \tau$ 
30:  end while
31: end procedure

```

SDM over ODM with several models, and that the sorting overhead is so low that it is likely to be less costly than the pre-simulation overhead of ODM [27].

2.4.7 Logarithmic Direct Method

In 2006, Li & Petzold released an unpublished manuscript describing Logarithmic Direct Method (LDM, Algorithm 8). This improvement is similar to ODM and SDM in the sense that it alters the average search depth during reaction selection, but by a different method. LDM performs a binary search on reaction propensities during reaction selection and can therefore claim to have $O(\log M)$ performance during this

step of the algorithm. It achieves this by first summing reactions propensities cumulatively (thus avoiding a sort), and performing the binary search on the cumulative reaction propensity array.

Algorithm 8 Logarithmic Direct Method (LDM) [28]

```

1: procedure LDM(molecular species, reactions)
2:   set simulation time  $t \leftarrow 0.0$  ▷ initialise
3:   initialise species state vector  $X$  [1.. $N$ ]
4:   store vector of reactions  $R$  [1.. $M$ ]
5:   initialise dependency graph  $DG$ 
6:   while  $t < t_{end}$  do
7:      $CalculatePropensities()$  ▷  $CalculatePropensities()$  described in Algorithm 6
8:     create cumulative sum array  $C$  [1.. $M$ ] ▷ select reaction  $\mu$  to fire
9:     set total propensity  $a_0 \leftarrow 0.0$ 
10:    for  $j \leftarrow 1$  to  $M$  do
11:       $a_0 += a_j$ 
12:       $C_j = a_0$ 
13:    end for
14:    generate  $r_1 \leftarrow rand()$ 
15:    target propensity  $a_t \leftarrow a_0 r_1$ 
16:     $\mu \leftarrow \text{binarysearch}$  for  $a_t$  in  $C$ 
17:    update state vector  $X \leftarrow X + R_\mu$  ▷ execute reaction
18:    generate  $r_2 \leftarrow rand()$  ▷ calculate reaction time
19:     $\tau \leftarrow -1.0 * \ln(r_2) / a_0$ 
20:    update simulation time  $t \leftarrow t + \tau$ 
21:  end while
22: end procedure

```

In the performance comparison present in the manuscript, LDM is consistently shown to significantly outperform ODM and SDM for several models. However, this finding is directly contradicted by Gillespie who states that LDM is slightly slower than ODM and SDM [7]. Also, the results show that ODM consistently slightly outperforms SDM, which is in opposition to the findings of McCollum et al. These contradictions lend strong support to the creation of a standardised benchmark of stochastic simulation algorithms in order to determine the most performant algorithm without the fear of bias, leading to my decision to create a benchmarking suite.

It should be noted that Gillespie states that due to numerical truncation, it would be maximally accurate to sort reaction indexes such that lowest reaction propensities occupy the lowest reaction indexes. This is the opposite ordering to that obtained with ODM or SDM, but LDM would be unaffected by any potential reordering as it performs a divide and conquer search on a cumulative array and thus removes the

effect of any pre-ordering. This is significant, as truncation in multi-scale networks with many order of magnitude differences between propensities may result in the lowest reaction *never firing* [7].

2.4.8 Partial Propensity Direct Method

Algorithm 9 Partial Propensity Direct Method (PDM) [29]

```

1: procedure PDM(molecular species, reactions)
2:   set simulation time  $t \leftarrow 0.0$  ▷ initialise
3:   initialise species state vector  $X$   $[1..N]$ 
4:   store vector of reactions  $R$   $[1..M]$ 
5:   init data structures  $n, \Pi, \Sigma, \Lambda, L, U^{(1)}, U^{(2)}, U^{(3)}$ 
6:   while  $t < t_{end}$  do
7:     generate  $r_1 \leftarrow rand()$  ▷ reacton selection
8:      $I, J = GetIndexesIandJ(r_1)$ 
9:      $\mu \leftarrow L_{I,J}$ 
10:    generate  $r_2 \leftarrow rand()$  ▷ calculate reaction time
11:     $\tau \leftarrow -1.0 * \ln(r_2) / a_0$ 
12:    update state vector  $X \leftarrow X + R_\mu$  ▷ execute reaction
13:    for all  $k$  in  $U_\mu^{(1)}$  do ▷ Update  $\Pi, \Sigma, \Lambda$  and compute  $\Delta a$ 
14:       $l \leftarrow U_{\mu,k}^{(1)}$ 
15:      for all  $m$  in  $U_l^{(3)}$  do
16:         $(i_m^l, j_m^l) \leftarrow U_{l,m}^{(3)}$ 
17:        if  $l \neq i_m^l$  then ▷ 5.2.2
18:           $\Pi_{i_m^l, j_m^l} \leftarrow \Pi_{i_m^l, j_m^l} + c_{\mu'}, \mu' = L_{i_m^l, j_m^l}$ 
19:        end if
20:        if  $l = i_m^l$  then
21:           $\Pi_{i_m^l, j_m^l} \leftarrow \Pi_{i_m^l, j_m^l} + \frac{1}{2} c_{\mu'}, \mu' = L_{i_m^l, j_m^l}$ 
22:        end if
23:        if  $l \neq j_m^l$  then ▷ 5.2.3
24:           $\Lambda_{i_m^l} \leftarrow \Lambda_{i_m^l} + c_{\mu'}, \mu' = L_{i_m^l, j_m^l}$ 
25:        end if
26:        if  $l = j_m^l$  then
27:           $\Lambda_{i_m^l} \leftarrow \Lambda_{i_m^l} + \frac{1}{2} c_{\mu'}, \mu' = L_{i_m^l, j_m^l}$ 
28:        end if
29:         $\Sigma_{temp} \leftarrow \Sigma_{i_m^l}$ 
30:         $\Sigma_{i_m^l} \leftarrow n_{i_m^l} \Lambda_{i_m^l}$ 
31:         $\Delta a \leftarrow \Delta a + \Sigma_{i_m^l} - \Sigma_{temp}$ 
32:      end for
33:       $\Delta a \leftarrow \Delta a + n_l \Lambda_l - \Sigma_l; \Sigma_l \leftarrow n_l \Lambda_l$ 
34:    end for
35:    update  $a \leftarrow a + \Delta a$ 
36:    update simulation time  $t \leftarrow t + \tau$ 
37:  end while
38: end procedure

```

The Partial Propensity Direct Method (PDM, Algorithm 9) introduced in 2009 by Ramaswamy et al, is unique in the sense that it scales with the number of species N in

the system rather than the number of reactions M which is likely to be much greater. They also introduced SPDM, a sorting version (with inspiration from SDM) which is particularly appropriate for stiff systems [29]. In other exact SSAs, computational efficiency may scale logarithmically or even in constant time with reactions in weakly coupled networks, however for strongly coupled networks these algorithms will still scale linearly. In strongly coupled networks the degree of coupling increases with system size and may even be as high as the number of reactions. With a high degree of coupling in the network, there will tend to be fewer species than there are in a weakly coupled network with the same number of reactions, because a high degree of coupling infers more reactions with shared reactants and products. In such a situation it is advantageous to scale with species rather than reactions [29].

The idea behind PDM is to *factor out* a particular species from each reaction, generating *partial propensities* that depend on the population of zero or one species. Ramaswamy et al. define “the partial propensity of a reaction with respect to one of its reactants as the propensity per molecule of this reactant” [29]. PDM uses novel data structures to update partial propensities including an implicit species dependency graph, but retains the same time step sampling method as DM.

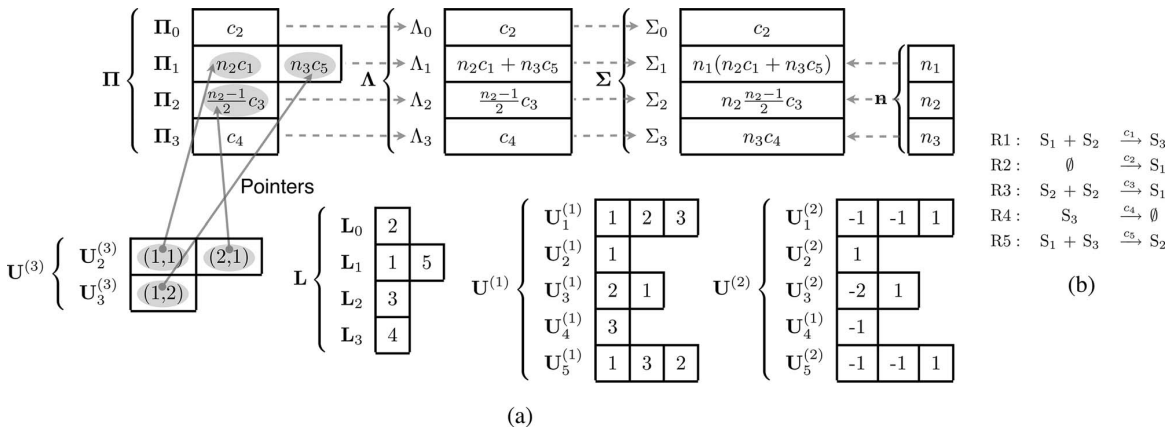


FIGURE 2.2: (a) Partial propensity direct method data structures
(b) Example reactions that populate data structures (taken from [29])

I recommend the original paper [29] as a comprehensive guide to implementing this algorithm, especially in regards to data structure implementation. However, due to

typographical errata in the original paper, corrections are required to steps 5.2.2 and 5.2.3 as detailed in Algorithm 9.

2.4.9 Composition Rejection

Algorithm 10 Composition Rejection Direct Method (CR) [30]

```

1: procedure CR(molecular species, reactions)
2:   set simulation time  $t \leftarrow 0.0$  ▷ initialise
3:   initialise species state vector  $X$  [1.. $N$ ]
4:   store vector of reactions  $R$  [1.. $M$ ]
5:   initialise dependency graph  $DG$ 
6:   set total propensity  $a_0 \leftarrow 0.0$ 
7:   set fixed number of groups  $G$ 
8:    $BuildGroups()$  ▷ init groups, calc all  $a_j$ ,  $a_0$ ,  $pmin$ 
9:   while  $t < t_{end}$  do
10:    if NOT first iteration then ▷ calculate reaction propensities
11:      for  $dep$  in  $DG_\mu$  do
12:         $a_0 -= a_{dep}$ 
13:        calculate updated propensity  $a_{dep}$ 
14:         $a_0 += a_{dep}$ 
15:      end for
16:       $rg \leftarrow CalcUpdatedReactionGroup(a_{dep})$ 
17:      if  $ReactionGroupUnchanged(rg)$  then
18:         $UpdateReactionGroupPropensity(rg)$ 
19:         $UpdateGroupPmax(rg)$ 
20:      else
21:         $RemoveFromReactionGroup(a_{dep})$ 
22:         $UpdateGroupPmax(rg_{remove})$ 
23:         $AddToReactionGroup(a_{dep}, rg)$ 
24:         $UpdateGroupPmax(rg)$ 
25:      end if
26:    end if
27:    generate  $r_1 \leftarrow rand()$  ▷ select reaction  $\mu$  to fire
28:    target propensity  $a_t \leftarrow a_0 r_1$ 
29:    create cumulative sum array  $C$  [1.. $G$ ]
30:     $SumGroupPropensitiesCumulativeArray(C)$ 
31:     $g \leftarrow \text{binarysearch for } a_t \text{ in } C$  ▷ find target group  $g$  for reaction  $\mu$ 
32:     $\mu = Rejection(g)$  ▷  $Rejection(g)$  described in Algorithm 11
33:    update state vector  $X \leftarrow X + R_\mu$  ▷ execute reaction
34:    generate  $r_2 \leftarrow rand()$  ▷ calculate reaction time
35:     $\tau \leftarrow -1.0 * \ln(r_2) / a_0$ 
36:    update simulation time  $t \leftarrow t + \tau$ 
37:  end while
38: end procedure

```

Slepoy et al introduced the Composition Rejection variant of DM (CR, Algorithm 10) in 2008, which claims to have constant time scaling $O(1)$, independent of M [30]. To achieve this, both reaction selection and propensity updates must be $O(1)$. Reaction

selection is performed via *rejection sampling* [30] which makes reaction selection independent of M . To imagine rejection sampling, consider a histogram with reactions on the x-axis and propensities on the y-axis. Two uniform random numbers are selected, the first one selects a reaction μ from 1 to M in order to choose a reaction that may potentially fire. The second uniform random number selects a value a_{rej} between 0 and p_{max} (the highest reaction propensity in the system). If $a_\mu \geq a_{rej}$, then the reaction is selected. This can be visualised as r being within the area covered by the propensity of reaction i in the plot. If $a_\mu < a_{rej}$ then the reaction is *rejected* and the algorithm is repeated until a reaction is selected [30].

Whilst rejection sampling makes reaction selection independent of M , it has an intrinsic cost, needing two random numbers to select a reaction instead of one, and also if there are many reactions rejected, this random number cost is repeated multiple times per iteration. To address this, the *composition* aspect of the CR is adopted. This simply means that reactions with similar propensity values are grouped together. Reactions are placed in groups from p_{min} to p_{max} where each group boundary is a cascading factor of 2 multiple of p_{min} . Arranging groups in this manner means that selecting a reaction from a particular group by parametrising the group's p_{max} , significantly reduces the number of rejections. This has an associated cost and precautions must be taken to maintain the $O(1)$ scaling. A third random number is needed to select the reaction group to fire, which is achieved by parametrising the total propensity and selecting by each group's total propensity, in a similar way to standard DM reaction selection. Figure 2.3 elucidates the composition grouping and rejection sampling mechanisms employed by the algorithm.

I must make G , the number of groups, independent of M to maintain $O(1)$ scaling by bounding its value. Slepoy et al. argue that if there is a reaction propensity distribution that requires a large number of groups, because of exponential group boundaries, one can postulate that reaction propensities under a certain value are so unlikely to fire they can be ignored and p_{min} increased. Whilst this fair assumption allows the generation aspect of CR to be $O(1)$, it can be argued that should this

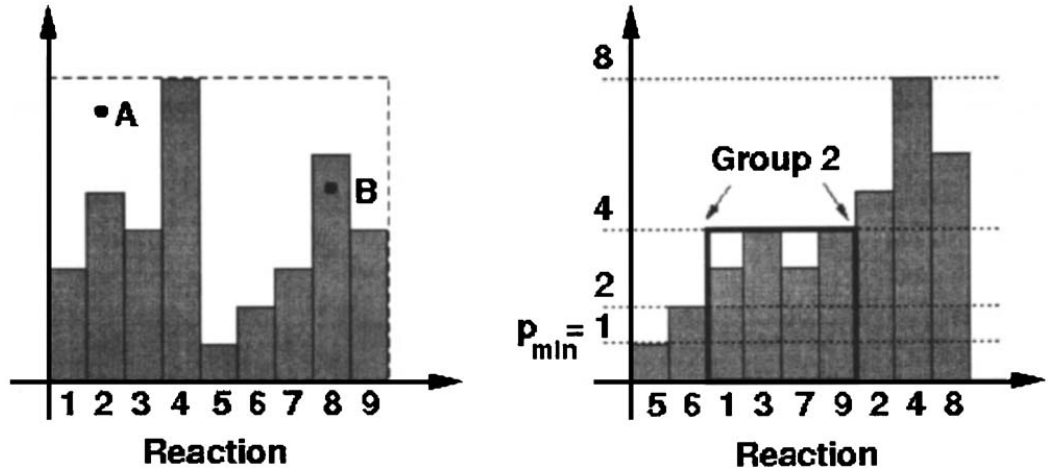


FIGURE 2.3: Composition and rejection algorithm for random variate generation. A reaction is selected from a set of reaction propensities left by picking random points A and B from a bounding rectangle until a point inside a vertical bar B is found. Grouping the propensities by their magnitude (right side sub-figure) makes rejected points less likely (taken from [30]).

situation arise, ignoring some reaction propensities would logically imply that this algorithm has become approximate rather than exact.

Algorithm 11 Rejection(g)

```

1: function REJECTION(group index g)
2:   while true do
3:     gs = num reactions in group
4:      $\mu = randInt(gs)$ 
5:      $a_{rej} = rand() * p_{max}$ 
6:     if  $a_{\mu} \geq a_{rej}$  then return  $\mu$ 
7:     end if
8:   end while
9: end function

```

For the update step of the CR algorithm, the dependency graph (from NRM) is used in order to determine which reactions are updated, and those are assigned to new groups if necessary. As changing the group a reaction belongs to can be performed in constant time, the authors state that the update aspect of the CR algorithm is $O(1)$. However, it should be noted that because part of the update step is updating the reaction propensities, it would be unwise to ignore the $O(\log M)$ scaling unless the reaction network is very weakly coupled. This can be highlighted in their results which demonstrate that the massive performance increase for CR over NRM for large reaction networks is mainly provided by the generation step, whilst the update step performance is similar to NRM. It should also be noted that because of CR's high

standing cost, algorithms with lower data structure overheads and consumption of random numbers will strongly outperform CR on smaller reaction networks.

2.4.10 Tau Leaping

Algorithm 12 Tau Leaping (TL) [11]

```

1: procedure TL(molecular species, reactions)
2:   SSA run counter  $g \leftarrow 0$  ▷ initialise
3:   set simulation time  $t \leftarrow 0.0$ 
4:   while  $t < t_{end}$  do
5:     if  $g > 0$  then
6:       run standard DM for  $g$  iterations
7:        $g \leftarrow 0$ 
8:     else
9:       if TauLeapStep() then ▷ TauLeapStep() described in Algorithm 13
10:      if  $g > 0$  then
11:        run standard DM for  $g$  iterations
12:         $g \leftarrow 0$ 
13:      end if
14:    else
15:      end simulation
16:    end if
17:  end if
18: end while
19: end procedure

```

τ leaping (TL, Algorithm 12) is a method created by Gillespie which first appeared in 2001 [11]. This method is distinct to the other methods previously mentioned in the sense that it is an *approximate* algorithm as opposed to an exact formulation. Gillespie noted that a strength of the exact SSA was that it meticulously considered every reaction occurring in the system, though performance was a caveat. Much of the detail of the exact simulations may be unnecessary and irrelevant to achieving an accurate picture of the system's temporal evolution, and these laborious simulations come at high computational expense. It would therefore be preferable to increment the temporal evolution of the system by a small but significant time interval each step rather than an infinitesimal τ , as long as acceptable accuracy could be achieved. Applying many individual reaction events in one algorithmic step (as opposed to one reaction per step) should provide a significant improvement to the time required to perform a simulation.

Algorithm 13 TauLeapStep()

```

1: function TAULEAPSTEP
2:   set restart leap flag  $rf \leftarrow false$ 
3:   set total propensity  $a_0 \leftarrow 0.0$ 
4:   set non-critical tau  $\tau_{ncr} \leftarrow 0.0$ 
5:   while first leap or re-leaping do
6:     if  $rf \neq true$  then ▷ if this is not a re-leap..
7:        $CalculatePropensities()$ 
8:        $a_0 \leftarrow SumPropensities()$ 
9:       if  $a_0 \leq 0.0$  then return  $false$ 
10:      end if
11:       $IdentifyCriticalReactions()$  ▷ described in Algorithm 14
12:       $\tau_{ncr} \leftarrow CalculateTauNCR()$ 
13:    end if
14:    if  $\tau_{ncr} \neq \infty$  &&  $\tau_{ncr} < \tau_{min}$  then ▷ check if tau leap is above min value
15:       $g \leftarrow 100$  ▷ Use DM for  $g$  runs
16:      break
17:    end if
18:     $a_{0crit} \leftarrow SumCriticalPropensities()$  ▷ choose tau for critical reactions
19:     $\tau_{crit} \leftarrow \infty$ 
20:    if  $a_{0crit} \neq 0.0$  then
21:      generate  $r_1 \leftarrow rand()$ 
22:       $\tau_{crit} \leftarrow -1.0 * \ln(r_1) / a_{0crit}$ 
23:    end if
24:     $\tau \leftarrow \min(\tau_{crit}, \tau_{ncr})$  ▷ tau should be smallest of crit/non-crit
25:    if  $\tau == \infty$  then return  $false$ 
26:    end if
27:    ▷ sample poisson distribution to fire reactions in tau leap
28:    ▷ critical reactions fired by monte carlo
29:     $FireReactions()$ 
30:    if  $HasNegativesInStateVector()$  then ▷ re-leap if negatives in state vector
31:       $ResetStateVector()$ 
32:       $\tau_{ncr} \leftarrow \tau_{ncr} / 2.0$ 
33:       $rf \leftarrow true$ 
34:    else
35:       $t \leftarrow t + \tau$ 
36:       $rf \leftarrow false$ 
37:    end if
38:  end while
39:  return  $true$ 
40: end function

```

In order to perform a faithful approximation of the system's temporal evolution with τ leaping, the *leap condition* must be met. Gillespie defines the leap condition as the requirement for τ to be small enough such that each leap will not result in an appreciable change to the propensity of any reaction. He also states that the more compliant a τ leap is to the leap condition, the greater the accuracy of the simulation [11]. Thus, a balancing act ensues where τ needs to be large enough that a performance increase is achieved, but small enough that good accuracy is

maintained. If it transpires that the τ required to satisfy the leap condition is so small such that only a handful of reaction events fire each τ leap, it is preferable to fall back to using DM until simulation conditions change to accommodate a significant leap (Cao et al suggest 100 iterations of DM before returning to τ leap [46]). The rationale being that there would be no efficiency advantage to using τ leaping in this situation; in fact the overheads of the τ leaping would result in the algorithm performing worse than an exact SSA. Moreover, if the exact SSA outperforms the approximation it is preferable to get an exact result.

If the leap condition is satisfied, the assumption is made that the propensity of each reaction channel is *constant* during the leap [11]. This assumption allows one to consider the probability of a particular reaction channel firing independently of other reaction channels. Therefore, one can sample the Poisson random variable with mean $a_j(x)\tau$ for each reaction channel to determine how many times each reaction has fired during a τ leap.

Gillespie acknowledged some issues that would need to be addressed upon introduction of his algorithm, namely an effective method to select the largest possible τ that satisfies the leap condition. Another issue was that it was possible for the state vector to end up with a negative species amount, an impossible situation that could not occur with an exact SSA formulation. This was addressed by Cao et al [47], who introduced a modified τ leap to avoid negatives occurring in the state vector. This is achieved by searching for *critical reactions* (see Algorithm 14), which are reactions that would result in a negative species amount after a pre-determined number of firings. Once critical reactions have been identified, the algorithm ensures that only one critical reaction can be fired in a time-step, whilst still “leaping” multiples of non-critical reactions as before.

In the rare case that negatives still occur in the state vector, the algorithm simply restarts the erroneous leap with a smaller value for τ . For the purposes of this thesis, I have implemented and considered the 2006 iteration of τ leaping with “efficient step size selection” [46]. This improves upon the original algorithm by increasing compliance with the leap condition whilst reducing the computational

Algorithm 14 IdentifyCriticalReactions()

```

1: function IDENTIFYCRITICALREACTIONS
2:   set critical reaction parameter  $K \leftarrow 10$ 
3:   set critical reaction flag  $CR [1..M] = \{false\}$ 
4:   for  $j \leftarrow 1$  to  $M$  do
5:     create temp copy of state vector  $X^{temp} \leftarrow X$ 
6:     for  $k \leftarrow 1$  to  $K$  do
7:        $X^{temp} \leftarrow X^{temp} + R_j$  ▷ apply reaction K times
8:     end for
9:     for  $i \leftarrow 1$  to  $N$  do
10:      if  $X_i^{temp} \leq 0$  then
11:         $CR_j \leftarrow true$ 
12:        break
13:      end if
14:    end for
15:  end for
16: end function

```

cost of determining the τ to leap. A major shortcoming of the original algorithm was that τ is bounded by a fraction of the sum of all reaction propensities. This means that in a multi-scale system, a particular reaction channel with a propensity orders of magnitude smaller than others may in fact be leaped with a τ that contravenes the leap condition for that reaction channel. The modified τ calculation considers the relative change in each reaction propensity in order to calculate the τ leap interval, rather than the absolute change of the sum of reaction propensities.

2.4.11 Reaction dependency graph (RDG)

The first SSA to use a dependency graph was Next Reaction Method (NRM) which was introduced in the year 2000 by Gibson & Bruck [25]. This algorithm used a *reaction dependency graph* (RDG) which is an interaction network that determines which reactions propensities need to be updated when a particular reaction is executed. More algorithms that used the RDG were subsequently introduced including Optimised Direct Method (ODM) [26], Sorting Direct Method (SDM) [27], Logarithmic Direct Method (LDM) [28] and Composition Rejection (CR) [30]. Whilst the RDG improved simulation times, there was a lack of discussion in the literature on the memory requirements and generation methods of this optimisation.

A reaction dependency graph is typically stored as a data structure that consists of a list of affected reaction indices for each reaction within a reaction network. Therefore, the worst case space complexity of the RDG is $O(M^2)$ (where M is the number of reactions in the network) and this occurs when the reaction network is fully coupled. Furthermore, the more coupled the network becomes, the more ineffective the RDG is at improving computational performance (as more propensities need to be recalculated per iteration). The naïve method of generating a reaction graph has time complexity of $O(M^2)$. This involves checking whether each reaction in the network affects any of the other reactions in the network.

Algorithm 15 Naive RDG Generation ($O(M^2)$)

```

1: procedure RDG(reactions)
2:   ▷ initialise
3:   store list of reactions  $RL$   $[1..M]$ 
4:   create empty dependency graph  $DG$ 
5:   for  $i \leftarrow RL[1..M]$  do
6:     for  $j \leftarrow RL[1..M]$  do
7:       if  $j$  is affected by  $i$  then
8:         ▷  $j$  is a dependency for  $i$ 
9:          $DG_i \leftarrow j$ 
10:      end if
11:    end for
12:  end for
13: end procedure

```

The memory and generation time requirements of the RDG have negative implications for exact stochastic simulation in fields such as Systems & Synthetic Biology, where reaction networks modelled grow in size with ever increasing biological knowledge. Consequently, when M is sufficiently large, generation times of the RDG will take longer than simulating a model without the RDG. Also when M is large, the RDG becomes intractable in terms of memory requirements, and high memory usage affects computational performance due to cache misses and memory paging [48]. The authors of the CR algorithm (which was formulated for large reaction networks) include analysis of a *low memory version* of their algorithm (i.e. no RDG) for this reason [30].

2.5 Modelling genetic biochemical systems

2.5.1 Modelling synthetic genetic logic gates

To demonstrate how the genetic regulatory machinery can be modelled and simulated, I shall explore an initial example biosystem. This exemplar system involves implementing Boolean logic gates in a gene regulatory context and produces a biochemical output based on the presence of biochemical inputs.

Synthetic Boolean logic gates have been addressed in various studies [49–51] and are of interest as the fundamental building blocks of potential biological computing. The devices discussed in this section are constructed using the genetic subcomponents of the XOR gate designed by Beal et. al [49]. Here, I consider two important logic gates: AND & OR. Both gates use two inducers, aTc and IPTG, as inputs. Inducers are chemicals that inhibit the activity of repressor transcription factors (i.e. they reduce the impact of transcription downregulation). aTc and IPTG inhibit the activities of TetR and LacI proteins, respectively. Both gates have green fluorescent protein (GFP) as an output to indicate a true/on resultant state for the system when production of GFP is high. The genetic designs of the gates are presented in Figures 2.4 & 2.5.

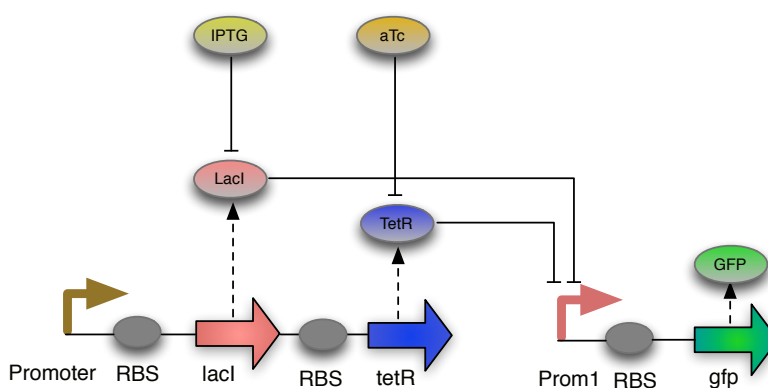


FIGURE 2.4: Genetic device functioning as an AND gate. Inputs to the gates are the molecular species IPTG and aTc. The output of the gate is given by the expressed amount of GFP molecules.

Figure 2.4 illustrates a genetic AND gate, which receives two input signals: aTc and IPTG. In this system, the transcription factors LacI and TetR are expressed by genes controlled by a single promoter. The aTc and IPTG molecules bind to TetR and LacI, respectively, to prevent them from inhibiting the production of GFP by binding to the corresponding promoter which up-regulates the expression of GFP. If both IPTG and aTc are set to high, then neither LacI nor TetR can inhibit GFP production and thus GFP production will be high.

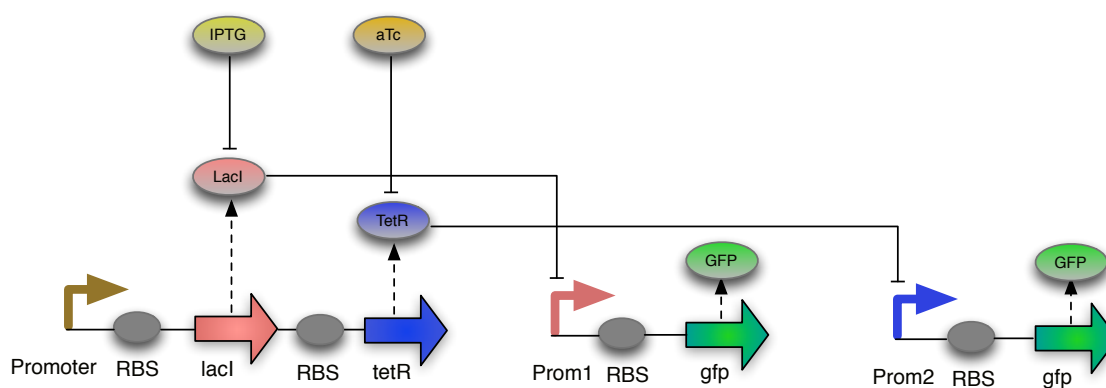


FIGURE 2.5: Genetic device functioning as an OR gate. Inputs to the gates are the molecular species IPTG and aTc. The output of the gate is given by the expressed amount of GFP molecules.

Figure 2.5 illustrates a genetic OR gate, comprising two separate mechanisms for inducing GFP production. Each mechanism has a unique promoter for each of the two GFP genes present in the system, allowing for individual activation of either GFP gene. As with the genetic AND gate, IPTG and aTc are used as inputs for the genetic OR gate. The production of GFP in the first mechanism is repressed by LacI whilst the second is repressed by TetR. As in the AND gate IPTG and aTc regulate LacI and TetR respectively. Because there are two separate GFP genes present controlled by two unique mechanisms, GFP can be produced when IPTG is set to high *or* when aTc is set to high (and when they are both set to high).

The stochastic model comprises a set of reaction channel rules governing the kinetic and stochastic behaviour of the system. Therefore, a modeller needs to convert the high level qualitative biological description shown in Figures 2.4 & 2.5 into a precise set of rules. Tables 2.4 & 2.5 present the rules and the kinetic constants of the

(a) AND gate

Rule		Kinetic constant
r_1 :	$\text{gene_LacI_TetR} \xrightarrow{k_1} \text{gene_LacI_TetR} + \text{mrna_LacI_TetR}$	$k_1 = 0.12$
r_2 :	$\text{mrna_LacI_TetR} \xrightarrow{k_2} \text{mrna_LacI_TetR} + \text{LacI}$	$k_2 = 0.1$
r_3 :	$\text{mrna_LacI_TetR} \xrightarrow{k_3} \text{mrna_LacI_TetR} + \text{TetR}$	$k_3 = 0.1$
r_4 :	$\text{LacI} + \text{IPTG} \xrightarrow{k_4} \text{LacI-IPTG}$	$k_4 = 1.0$
r_5 :	$\text{TetR} + \text{aTc} \xrightarrow{k_5} \text{TetR-aTc}$	$k_5 = 1.0$
r_{6a} :	$\text{gene_GFP} + \text{LacI} \xrightarrow{k_{6a}} \text{gene_GFP-LacI}$	$k_{6a} = 1.0$
r_{6b} :	$\text{gene_GFP-LacI} \xrightarrow{k_{6b}} \text{gene_GFP} + \text{LacI}$	$k_{6b} = 0.01$
r_{7a} :	$\text{gene_GFP} + \text{TetR} \xrightarrow{k_{7a}} \text{gene_GFP-TetR}$	$k_{7a} = 1.0$
r_{7b} :	$\text{gene_GFP-TetR} \xrightarrow{k_{7b}} \text{gene_GFP} + \text{TetR}$	$k_{7b} = 0.01$
r_8 :	$\text{gene_GFP} \xrightarrow{k_8} \text{gene_GFP} + \text{GFP}$	$k_8 = 1.0$
r_9 :	$\text{GFP} \xrightarrow{k_9}$	$k_9 = 0.001$
r_{10} :	$\text{LacI} \xrightarrow{k_{10}}$	$k_{10} = 0.01$
r_{11} :	$\text{TetR} \xrightarrow{k_{11}}$	$k_{11} = 0.01$
r_{12} :	$\text{mrna_LacI_TetR} \xrightarrow{k_{12}}$	$k_{12} = 0.001$

TABLE 2.4: Kinetic rules for the Boolean AND gate.

(b) OR gate

Rule		Kinetic constant
$r_1 - r_5$	same as the rules $r_1 - r_5$ of the AND gate	
r_{6a} :	$\text{gene_GFP1} + \text{LacI} \xrightarrow{k_{6a}} \text{gene_GFP1-LacI}$	$k_{6a} = 1.0$
r_{6b} :	$\text{gene_GFP1-LacI} \xrightarrow{k_{6b}} \text{gene_GFP1} + \text{LacI}$	$k_{6b} = 0.01$
r_{7a} :	$\text{gene_GFP2} + \text{TetR} \xrightarrow{k_{7a}} \text{gene_GFP2-TetR}$	$k_{7a} = 1.0$
r_{7b} :	$\text{gene_GFP2-TetR} \xrightarrow{k_{7b}} \text{gene_GFP2} + \text{TetR}$	$k_{7b} = 0.01$
r_8 :	$\text{gene_GFP1} \xrightarrow{k_8} \text{gene_GFP1} + \text{GFP}$	$k_8 = 1.0$
r_9 :	$\text{gene_GFP2} \xrightarrow{k_9} \text{gene_GFP2} + \text{GFP}$	$k_9 = 1.0$
$r_{10} - r_{13}$	same as the rules $r_9 - r_{12}$ of the AND gate	

TABLE 2.5: Kinetic rules for the Boolean OR gate.

devices described above. If one considers the AND gate, Rules r_1 to r_3 describe the expression the LacI and TetR proteins from `gene_LacI_TetR`, regulated by the same promoter. Rules r_4 and r_5 describe the binding of LacI to IPTG and TetR to aTc, respectively. Rules r_{6a} and r_{6b} describe the inhibition activity of LacI, i.e. its binding to the promoter that upregulates the GFP production. Rules r_{7a} and r_{7b} define the

same process for TetR. Rule r_8 describes the expression of GFP. Rules r_9 to r_{12} define the degradation process of various molecular species. The input molecules aTc and IPTG are kept constant in the model to stop them being quickly consumed and thus maintain a persistent output state in the model.

When considering this model and the respective stochastic rules, it should be noted that there is no consideration for biological intermediates such as mRNA production. Furthermore, whilst the RBS is shown in the Figures for the gates, it is overlooked for the stochastic rules. Whilst these entities could indeed be considered in the stochastic rules, a decision is made by the modeller regarding the level of detail that is required. In this particular system, the behaviour that I wish to observe (Boolean gate mechanics) should be captured at this scale. If the simulations performed on this model do not match hypotheses or biological reality, it may then be necessary to increase detail level until the desired behaviour is captured by the model.

2.5.1.1 Systems Biology Markup Language (SBML)

Systems Biology Markup Language (SBML) is a model format created to standardise the description of biochemical models [33]. The stated motivation of the format was to allow biological models to be “*shared, evaluated and developed cooperatively*”. SBML is touted as a “*software independent language*”, enabling interoperability between different modelling and simulation platforms. SBML is an XML based format that is supported by many different frameworks. A free software library is in continuous development for the SBML standard, called *libSBML* [52]. This library supports many different programming languages and handles the parsing of SBML models for developers. The SBML model standard contains a comprehensive set of formalised biological modelling types and operators, including those that are essential to stochastic simulation: *species*, *rules* (i.e. reactions), and *parameters* (i.e. stochastic rate parameters). Other more advanced features that can be employed by stochastic modellers include *events* and *compartments*. Compartments provide a level of spatial resolution as well as *separation* which is analogous to a cell membrane.

2.5.2 Simulating synthetic biological boolean gates

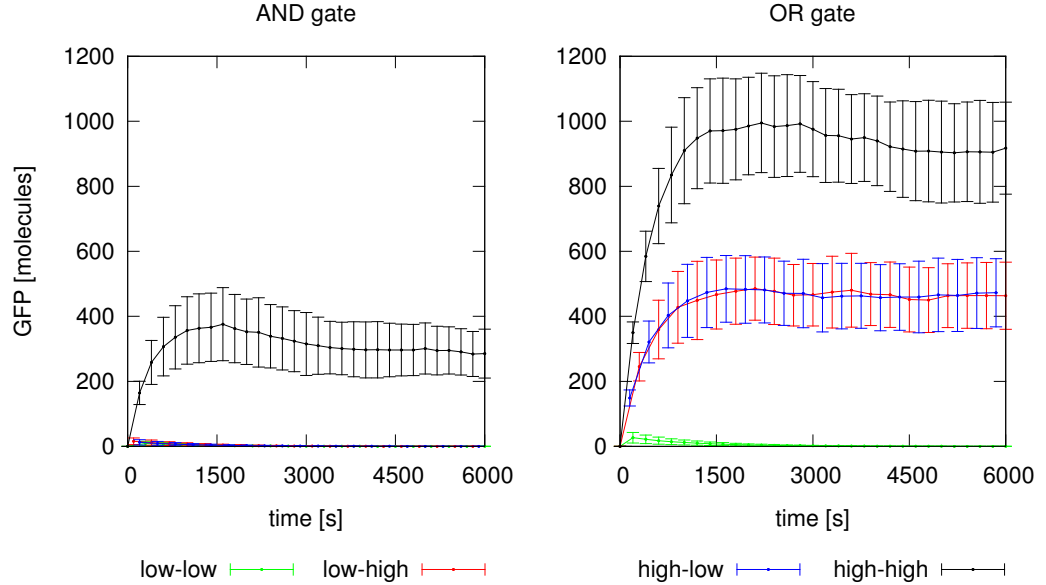


FIGURE 2.6: GFP expression in the AND (left) & OR gates over time for the aTc/IPTG input combinations *low-low*, *low-high*, *high-low*, and *high-high*. Error bars denote the standard deviations of 100 statistically independent samples.

Simulation of the stochastic models detailed in Section 2.5.1 is performed using the Gillespie SSA [10, 31]. At each reaction execution, the system state vector of molecular species is adjusted and a time-series trajectory of the system can be logged. To perform simulations of the models, I use my *ngss* (next generation stochastic simulator) software [24]. Ngss simulates stochastic models provided in SBML format [33] and generates time-series for all the molecular species present in the system. Time-series data is outputted and recorded as plain text comma separated values. For each model I tried four different configurations of gate inputs aTc and IPTG (*high-high*, *high-low*, *low-high* and *low-low*) where low is zero molecules and high is 1000 molecules.

Trajectories of both gate dynamics are shown in Figure 2.6 for the four different input combinations of low and high aTc and IPTG. The gates quickly approach a steady state with output concentrations that implement the desired Boolean logic. During the short transient period, GFP is produced in marginal quantities even in

the absence of input signals, but this expression is suppressed once LacI and TetR repress the respective promoters and the present GFP degrades.

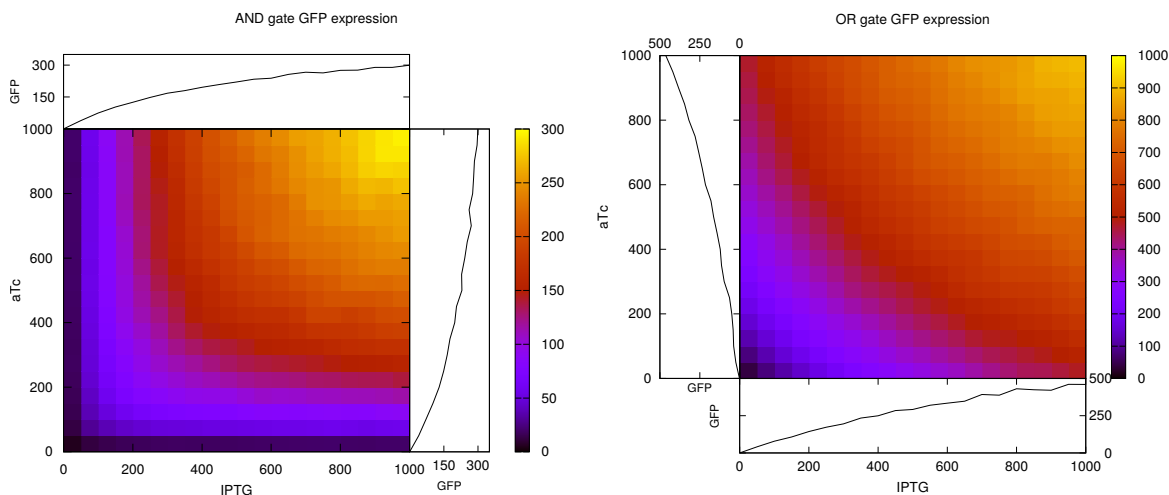


FIGURE 2.7: Heat map visualisations of the AND & OR gate transfer functions obtained by stochastic simulation. Colours indicate GFP expression for different aTc & IPTG input values. The top inlay shows the steady-state response of the gate for varying IPTG amounts under constant $aTc = 1000$, the right inlay shows the gate response for varying aTc under constant $IPTG = 1000$.

Figure 2.7 show the transfer functions (gate output for varying input values) of the AND and OR gates. In principle, the genetic AND and OR devices closely implement the requested transfer functions and express high GFP amounts under the presence of both (AND gate) or either of the two inputs (OR gate). Yet, the simulations also reveal that the gate outputs follow their inputs more or less linearly and do not implement a clear switching behaviour where the output concentration would drastically change around some critical threshold input value. Depending on the application, the observed linear behaviour can cause problems by accumulating errors when complicated circuits are composed by feeding the output of one gate into other downstream gates.

2.5.3 Benchmarking models

Ngss supports nine different variants of the SSA that each employ various optimisations in order to improve computational performance. Eight exact SSA formulations are included. These are Direct Method (DM) [31] and First Reaction Method (FRM)

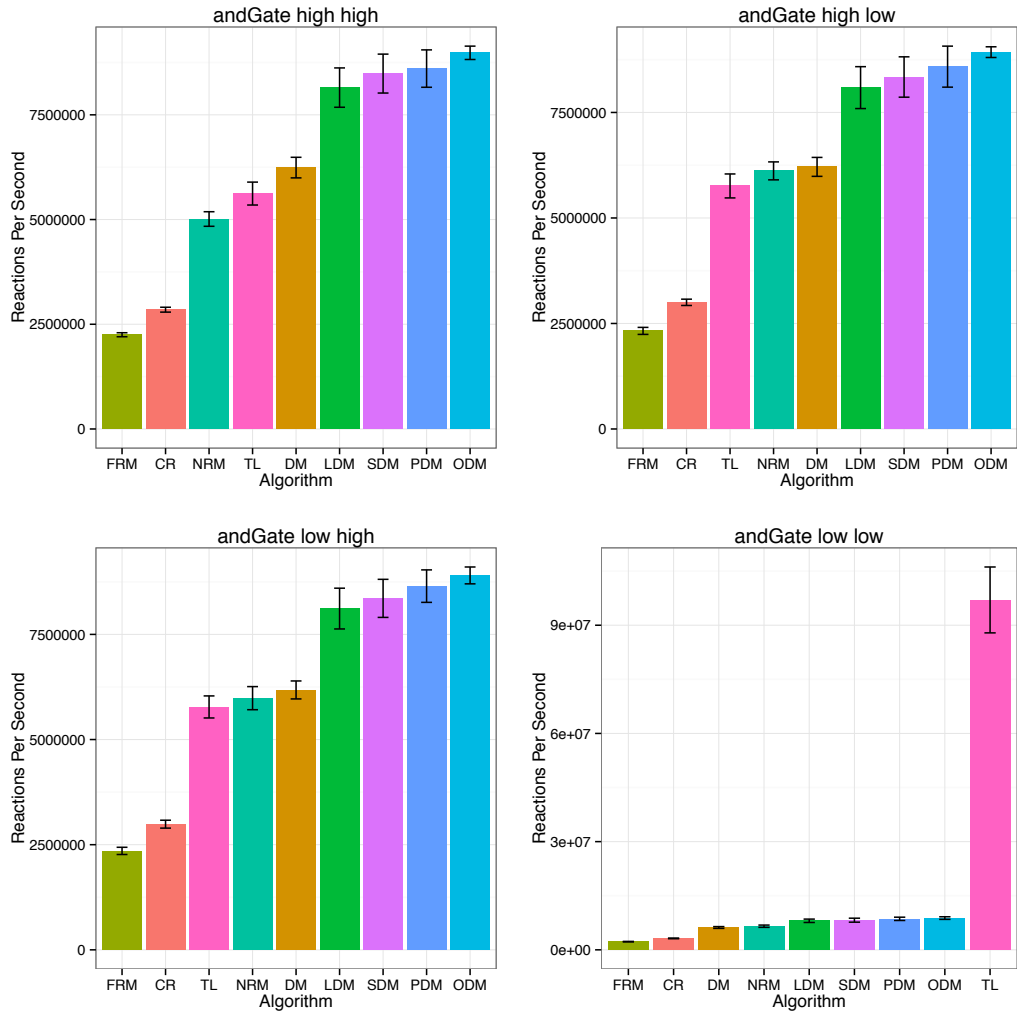


FIGURE 2.8: Algorithm benchmark performance results in *rps* (higher is better) of each algorithm for the AND gate with aTc and IPTG in *high-high* (constant 1000 1000) and *low-low* (constant 0 0) input configuration. Each algorithm's performance was evaluated as the mean of a total of 100 runs.

[10], Next Reaction Method (NRM) [25], Optimised Direct Method (ODM) [26], Sorting Direct Method (SDM) [27], Logarithmic Direct Method (LDM) [28], Partial Propensities Direct Method (PDM) [29] and Composition Rejection (CR)) [30]. An approximation algorithm, Tau Leaping (TL) [11] is also considered.

As I am concerned with improving the simulation time for a particular model, I benchmarked the performance of each of the mentioned SSA variants for the Boolean gate models. For each algorithm, 100 runs were performed and each simulation completed to 6000 seconds of simulation time. The metric for measuring performance

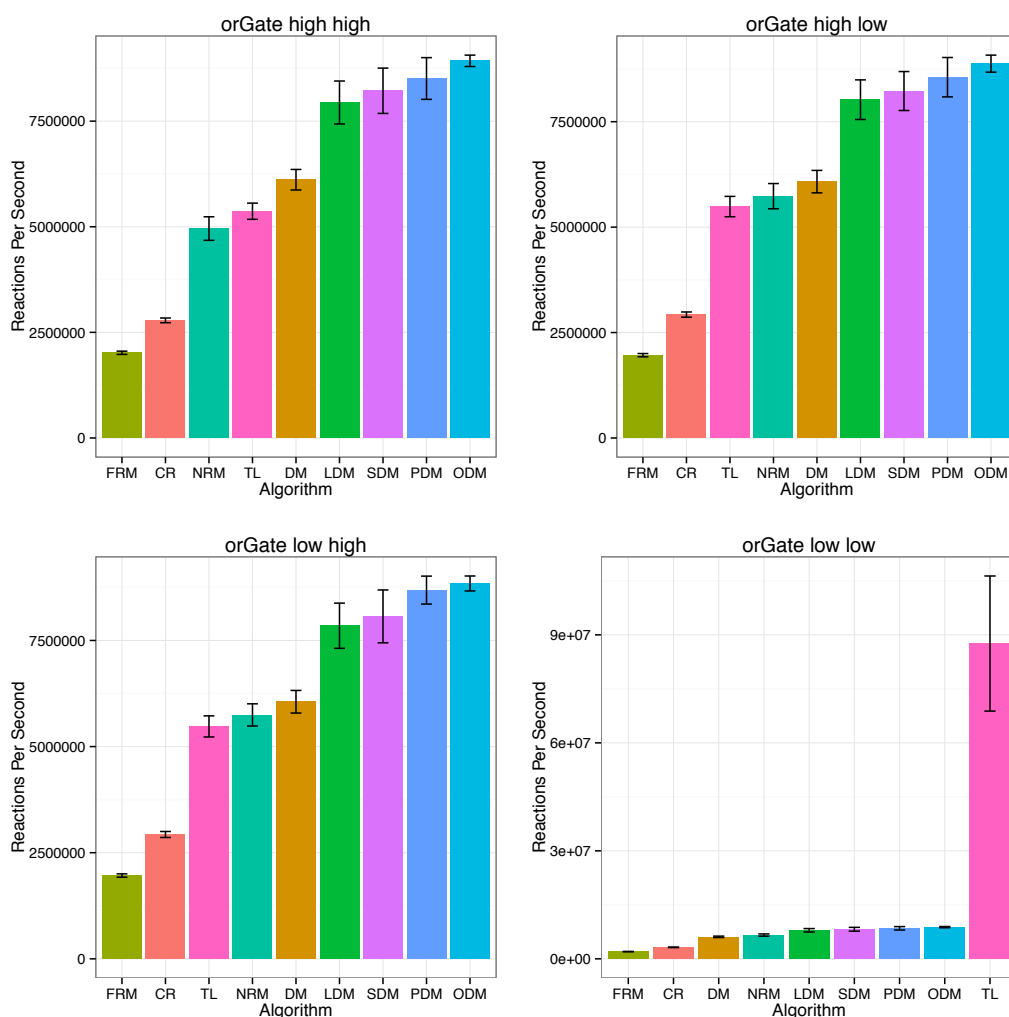


FIGURE 2.9: Algorithm benchmark performance results in *rps* (higher is better) of each algorithm for the OR gate with aTc and IPTG in *high-high* (constant 1000 1000) and *low-low* (constant 0 0) input configuration. Each algorithm's performance was evaluated as the mean of a total of 100 runs.

used is *reactions per second (rps)*. Rps is calculated by dividing the number of reactions executed by the amount of computational (process) time required. Whilst many comparative benchmarks use simpler metrics for measuring performance such as runtimes, this is not appropriate for measuring multiple samples of a stochastic simulation. Different runs of a stochastic biochemical model will generate a different stochastic trajectory before hitting a shared simulation end point, and the number of executed reactions and computational work done by each run might vary. Rps is therefore a more appropriate metric for algorithmic computational performance than runtime because it removes the effect of stochasticity on algorithmic runtime.

Rps is analogous to a *relative speed up* metric, where higher is better.

The algorithmic performance profiles of the different input combinations for both the OR gate and AND gate were similar, with identical algorithm performance rankings per input combination. Figures 2.8 and 2.9 show the algorithmic performance results (in reactions per second of CPU time) for the AND and OR gate models respectively. These results demonstrate that even very small differences in a model (in this case, the initial concentrations of two species) may result in large differences in algorithmic performance profiles. One can see that for the *low low* configuration TL is the fastest performing simulation algorithm, and outperforms others by an order of magnitude. However, in all other configurations ODM is the better algorithmic selection and strongly outperforms TL.

Chapter 3

Modelling and Stochastic Simulation of Biochemical Models

3.1 Introduction

Stochastic simulation for systems & synthetic biology requires an understanding of the fundamental biological constituents of complex biochemical systems. A biological concept must first be formulated of the target system, to be translated into a formalised model description that adheres to a format suitable for simulation. This chapter introduces some of the biological knowledge required for stochastic modelling and explores some biological systems from the literature. There is a discussion of how these biological systems can be modelled and simulated, followed by an initial benchmark of stochastic simulation performance. This preliminary analysis will elucidate the motivation for the first hypothesis of this thesis: *There is no single SSA that is superior in performance for every biomodel.*

“ To do this [*simulate biology*] effectively, not only must we use the vocabulary of the machine language, but we must also pay heed to what may be called the grammar of the biological system. ”

Sydney Brenner [53]

3.2 Simulating models from the literature

3.2.1 Experimental models

The first dataset consists of eight fully specified stochastic biological models (see Table 3.1). This dataset has been collected from the literature and curated, thus it will be referred to as the *curated models dataset* in this thesis. These models have been sourced from the literature concerned with the analysis of real biological systems and incorporate biologically plausible models of these systems. I intend for these models to represent a snapshot of “real world models” used by biologists.

Model	Description	Reference	Species	Reactions
A1	cAMP Oscillations	[54]	8	14
A2	Heat Shock Response	[55]	28	61
A3	E.Coli QS Circuit	[56]	22	25
A4	Thermodynamic Switch	[13]	16	24
A5	Auxin Transport	[12]	43	124
A6	G Protein Signalling	[57]	19	26
A7	Exponential Growth	[58]	200	920
A8	lacZ lacY Expression	[59]	23	22

TABLE 3.1: Summary of models available in the curated models dataset.

NOTE: A second and much larger dataset for this thesis is introduced in Section 4.2.2.

3.2.2 Model A1: Robust cAMP oscillations during *Dictyostelium* aggregation

The first model I shall explore investigates *cyclic adenosine monophosphate* (cAMP) in the social amoeba *Dictyostelium discoideum* [54]. cAMP is used as an intracellular signalling molecule for many different organisms. cAMP binds to the *cAMP protein receptor* (CRP) which is a transcription factor typically regulating many genes. For

example, in the bacterium *Escherichia coli*, CRP is involved in the mechanisms of over 50% of transcription units and affects up to 200 promoters [60].

During periods of starvation, *Dictyostelium* cells transition to an aggregation state in which they group together to produce spores. This process is regulated by cAMP, which *Dictyostelium* is able to secrete in order to induce the behaviour in surrounding cells. Stimulation of cAMP production occurs in an oscillatory fashion, with *Dictyostelium* generating pulses of cAMP following a regular period [54].



FIGURE 3.1: Aggregation of *Dictyostelium discoideum* (taken from [61]).

This system was initially modelled deterministically [62], producing the expected period and amplitude with the specified parameters [54]. However, *in vivo*, the system is subject to large fluctuations for its parameters. Analysis has shown this model is not robust in the face of minor changes to its parameter space [63]. This contradicts biological reality where the *Dictyostelium* cAMP oscillations remain robust in the face of large variation.

Figure 3.2 (A) shows the model evaluated by Kim et al. [54] which is based on the deterministic model from Laub and Loomis [62]. Kim et al. use a perturbed parameter set and simulate the model both deterministically and stochastically. Their simulation results demonstrate that an altered parameter set causes the deterministic model to terminate its oscillatory behaviour. However, they show that by using stochastic methods the model's oscillatory behaviour actually remains robust. These simulation results are shown in Figure 3.2 (B), with a blue deterministic simulation plot and a red stochastic simulation plot.

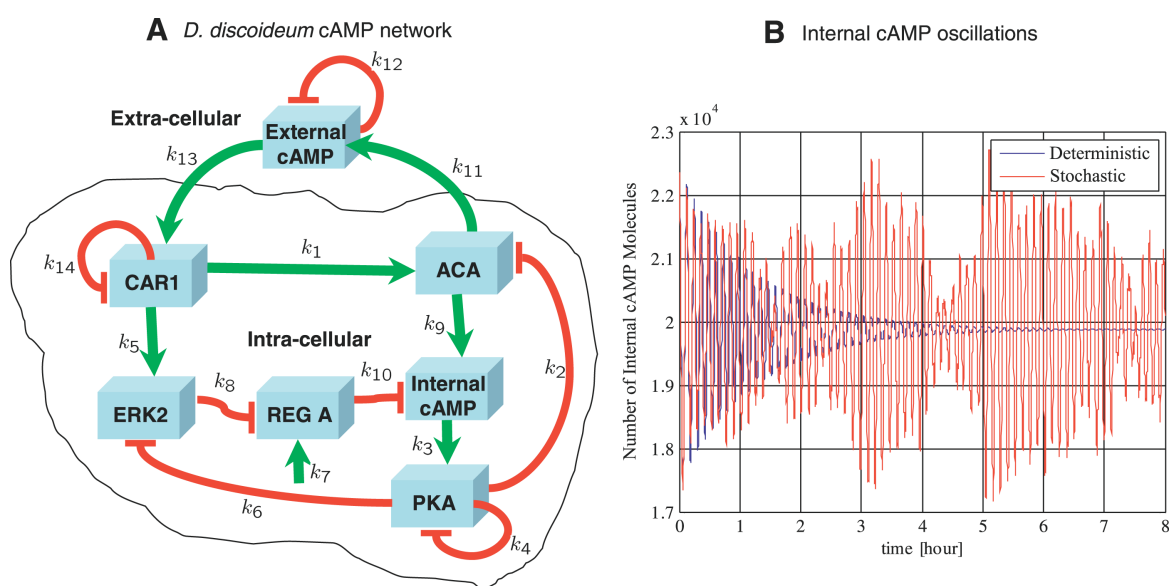


FIGURE 3.2: Gene regulatory model and simulation results of cAMP production during *Dictyostelium* aggregation (taken from [54]). Part (A) of this figure shows the gene regulatory network involved in *Dictyostelium* aggregation. Part (B) of this figure shows the results of simulations performed on the model. The blue plot shows the results of an ODE solver for this model. The red plot shows the results of a stochastic simulation for this model.

This result is significant because a biologist using a deterministic method of evaluating this model might conclude that the model is incorrect, whilst the issue lay with the simulation methodology itself. Kim et al. communicate that this type of oscillating model should be simulated stochastically and not deterministically, even with high species amounts. The rationale for SSA use is typically for models with low molecular species counts, but this particular model has relatively high species

amounts and yet is still heavily influenced by the effects of stochastic noise. The analysis indicates that stochastic noise is an important source of intracellular robustness [54].

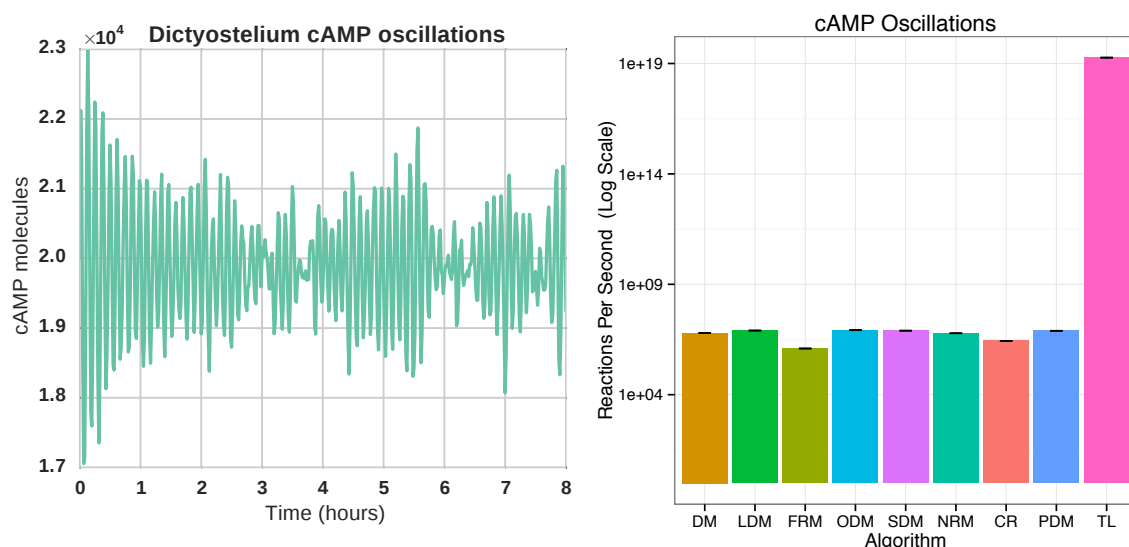


FIGURE 3.3: Stochastic simulation results for repeated cAMP oscillation experiment and SSA performance benchmark for nine different algorithms for model. The left hand side figure shows a repeated experiment of cAMP oscillations using the ngss simulator. This simulation was run using the Direct Method algorithm running for 480 minutes of simulation time. The right hand side figure shows the benchmark results for this model over nine algorithms. The benchmark was performed on a single core of an Intel i7 2600K with 16GB RAM. The benchmark was run on each algorithm-model combination with error bars showing the variation over 10 samples.

The left hand side of Figure 3.3 shows a repeat of the experiment performed by Kim et al. using the Direct Method algorithm from ngss simulation software to confirm the robust oscillatory behaviour. The same robust oscillatory behaviour was verified for all the nine SSA implementations of interest.

Figure 3.3 also shows the benchmark of the nine SSA implementations for the cAMP oscillation model. One can see that algorithmic performance is similar for all algorithms with the exception of TL. TL is able to apply multiple reactions per algorithmic step if there is only a small relative change in propensity for the system. This system has high levels of cAMP molecules which allows TL to apply multiple reactions involving cAMP molecules in a single step which drastically improves algorithmic

performance. It is quite clear from the benchmark results that a scientist would want to select TL for this model as it is orders of magnitude faster than other formulations.

When benchmarking a model that displays oscillatory behaviour, one might expect SDM to exhibit strong performance compared to other SSA formulations. This is because SDM dynamically sorts reactions (loosely by propensity) in order to reduce reaction search depth at each iteration. However, this model only has 14 reactions and of those only a few reactions involving cAMP will dominate the system. Thus the low numbers of reactions in this system means that the performance advantage from reduced reaction search depth is negligible.

3.2.3 Model A2: Heat shock response in *Escherichia coli*

Model A2 investigates the regulation of the *heat shock response* genetic circuit in *Escherichia coli* [55]. Organisms strive to maintain homeostasis and possess repair mechanisms to ensure biochemical robustness. Amongst these mechanisms, heat shock response is a gene regulatory subsystem that detects and repairs damage caused by heat shock, oxidative stress, toxins and other stressors [64].

Proteins in a cell that sustain heat shock unfold and *denature* (lose their precise protein structure). In response, heat shock proteins (HSPs) are produced that behave as *molecular chaperones* to help refold denatured proteins or as *proteases* to degrade them. There is an important balance to be maintained, as producing HSPs places a large metabolic burden on the organism. However, without HSPs, heat shock will disrupt normal cellular function.

Figure 3.4 shows an overview of the heat shock response model A2. Regulation of heat shock response is controlled by the *sigma factor* protein σ^{32} . A sigma factor is a transcription initiation factor which binds to RNAP to induce transcription of relevant genes, but dissociates prior to transcription [66]. Kurata et al. model important network motifs involved in the heat shock response [55]. Firstly, heat shock invokes a *feed-forward* motif by greatly increasing *rpoH* mRNA translation. Secondly, *feedback* motifs regulate the levels of σ^{32} to minimise the resultant metabolic costs. The

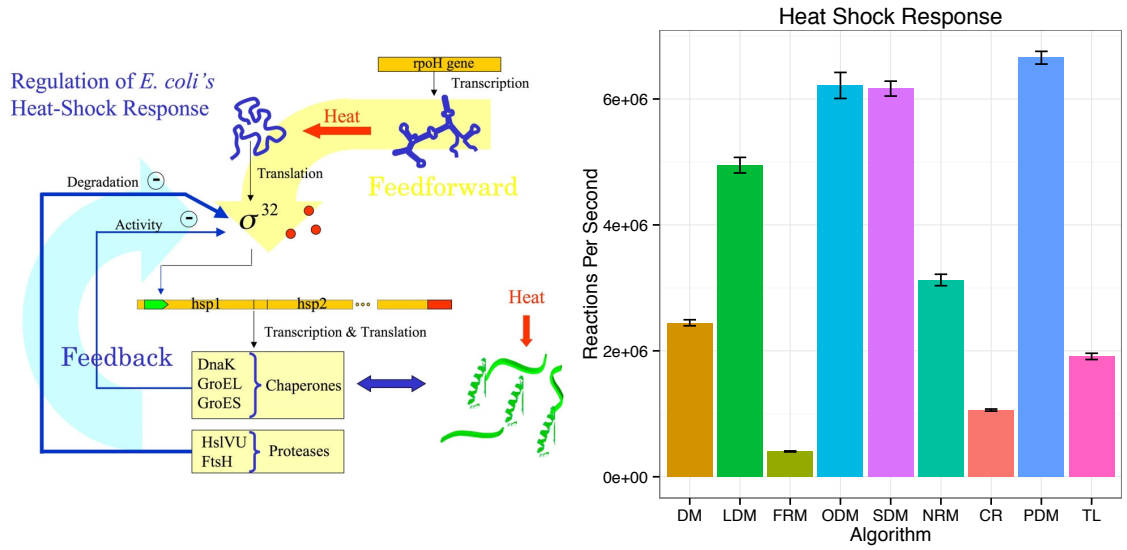


FIGURE 3.4: Model of heat shock response in *Escherichia coli* with SSA performance benchmark for nine different algorithms. The left hand side figure shows an overview of the heat shock response regulation model (taken from [65]). The right hand side figure shows the benchmark results for this model over nine algorithms. The benchmark was performed on a single core of an Intel i7 2600K with 16GB RAM. The benchmark was run on each algorithm-model combination with error bars showing the variation over 10 samples.

model shows σ^{32} upregulating the *FtsH* protease which degrades σ^{32} . The model also shows that the σ^{32} upregulated *DnaK* molecular chaperone will inhibit σ^{32} transcription initiation if *DnaK* is not involved in repair activity. This means that when HSPs complete their repairs, they terminate the heat shock response behaviour to reduce the metabolic load on the biosystem.

Kurata et al. claim that their model accurately reproduces heat shock behaviour even when tested experimentally with relevant gene knockout mutants [55]. They note that low numbers of σ^{32} molecules in the system may subject it to stochastic fluctuations in behaviour. Therefore, they performed stochastic simulations to complement their deterministic model of heat shock response. However, they found that the effects of stochastic noise did not alter heat shock response when compared to the deterministic model.

The right side of Figure 3.4 shows the results of the SSA performance benchmark for the heat shock response model. PDM, which claims superior performance for highly

coupled reaction networks, is the highest performing algorithm for this model closely followed by SDM and ODM. Reaction networks are considered “coupled” when the products of an arbitrary reaction are likely to affect the reactant populations of other reactions. The key features of this model involve feedback and feed-forward *loop* motif behaviours for σ^{32} , which implies that the network is highly coupled.

Comparing the benchmarks of model A1 in Figure 3.3 and model A2 in Figure 3.4, one can see a large difference in algorithm performance profiles. Most of this difference is explained by the very strong performance of TL in model A1. This is the first piece of evidence demonstrating large variations in SSA performance between models.

3.2.4 Model A3: *Escherichia coli* AI-2 quorum sensing circuit

Model A3 represents the genetic circuit regulating AI-2 quorum sensing in *Escherichia coli* (see Figure 3.5). Quorum sensing is a decentralised social mechanism employed by organisms such as bacteria to co-ordinate behaviour. Bacterial quorum sensing uses signalling molecules (autoinducers) which are secreted by individual cells. The autoinducers can then be “sensed” by neighbouring cells, thus quorum sensing be thought of a simple communication system. In small numbers, these signalling molecules do not induce a change in behaviour. However, when a threshold amount of autoinducers is “detected” by a cell, a shift in behaviour is induced. Quorum sensing can co-ordinate group behaviours as varied as biofilm formation, cell division, virulence and motility [56].

More specifically, autoinducer signalling molecules bind to receptors and induce gene expression. For example, an autoinducer can bind to a transcription factor protein and activate it causing an up-regulation of gene expression. In bacterial quorum sensing, gene expression involved in synthesis of the autoinducer is up regulated by itself causing a positive feedback loop. Model A3 evaluates the synthesis of AI-2 (Autoinducer 2) in *Escherichia coli* via the Pfs-LuxS pathway. AI-2 is a family of signalling molecules that are employed by many different species. Using this model,

Li et al. show that dramatically increased levels of AI-2 in the presence of glucose are not dependent on Pfs or LuxS levels. Therefore, their experiments show that an alternate (glucose regulated) AI-2 synthesis pathway must exist [56].

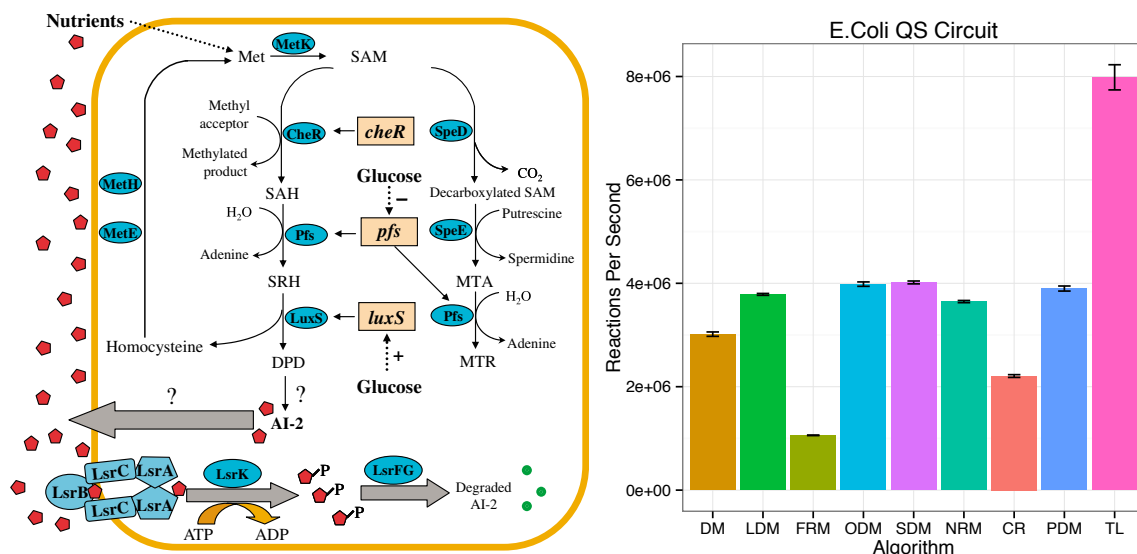
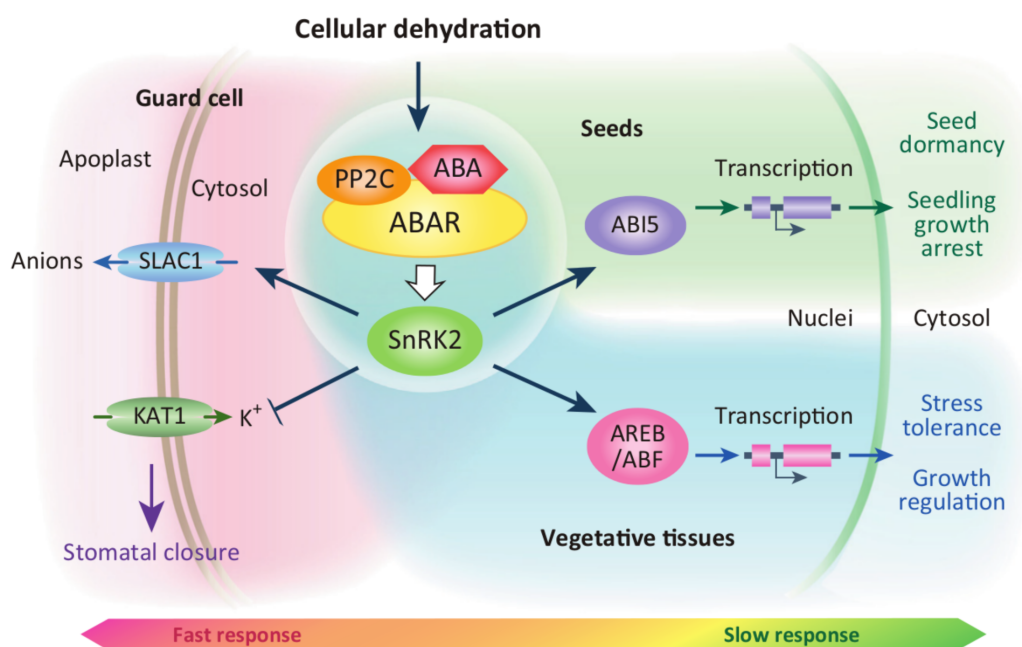


FIGURE 3.5: Model of AI-2 synthesis and uptake pathways in *Escherichia coli* with SSA performance benchmark for nine different algorithms. The left hand side figure shows AI-2 synthesis and uptake pathways in *Escherichia coli* (taken from [56]). The right hand side figure shows the benchmark results for this model over nine algorithms. The benchmark was performed on a single core of an Intel i7 2600K with 16GB RAM. The benchmark was run on each algorithm-model combination with error bars showing the variation over 10 samples.

Figure 3.5 shows the SSA performance benchmark for the Li et al. *Escherichia coli* model [56]. The SSA performance profile for this model is similar to model A1, if one compares Figures 3.5 and 3.3. However, whilst the results of model A1 are shown on the logarithmic scale because of the very high relative performance of TL, this model does not have such a large difference in performance between any algorithm. TL is also the fastest performing algorithm for this model, which is related to the high amounts for various species in the model. As the performance difference between TL and other algorithms is not by many orders of magnitude, this suggests that the algorithm does not “leap” at every algorithmic iteration or that leaps are for smaller time-steps. One should also note that the SSA performance profile appears different to that of model A2 because of the strong relative performance of TL, but the algorithm performance rankings are otherwise similar.

3.2.5 Model A4: Thermodynamic switch modulating abscisic acid receptor sensitivity

Abscissic acid (ABA) is a phytohormone that regulates growth when the plant is subject to environmental stressors such as drought, salinity and cold weather [67]. For example, during winter ABA inhibits cell division and slows plant growth. ABA can reduce transpiration during periods of dehydration by closure of stomata. ABA also regulates seed dormancy to ensure that seeds do not germinate during poor conditions for survival [68].



TRENDS in Plant Science

FIGURE 3.6: Major abscisic acid (ABA) signalling pathways in response to cellular dehydration (taken from [69]). ABA, ABA receptors (ABARs) and protein phosphatases 2C (PP2Cs) regulate sucrose non-fermenting-1 protein kinase 2 (SnRK2s) control both fast and slow ABA signalling pathways in response to cellular dehydration. Fast signalling can invoke stomatal closure, whereas slow pathways instigate transcriptional regulation of stress response genes.

ABA acts on 14 receptors in *Arabidopsis* of the PYR/PYL/RCAR protein family that regulate the behaviour of 2C-type protein phosphatases (PP2Cs) [70]. PP2Cs regulate the phosphorylation of sucrose non-fermenting-1 protein kinase 2 (SnRK2s), keeping them inactive during times of low environmental stress [13]. Phosphorylation is a reversible process that switches the activity enzymes or receptors on or off,

regulating cellular signalling and is essential to nearly every cellular process [71]. During periods of environmental stress, ABA levels increase and inhibit PP2C activity by inducing stable complexes between PP2Cs and PYR/PYL/RCAR. The reduced PP2C activity results in the presence of active SnRK2 kinases which enable stress response via phosphorylation. Figure 3.6 shows the signalling pathways controlled by the activity of the SnRK2 kinases.

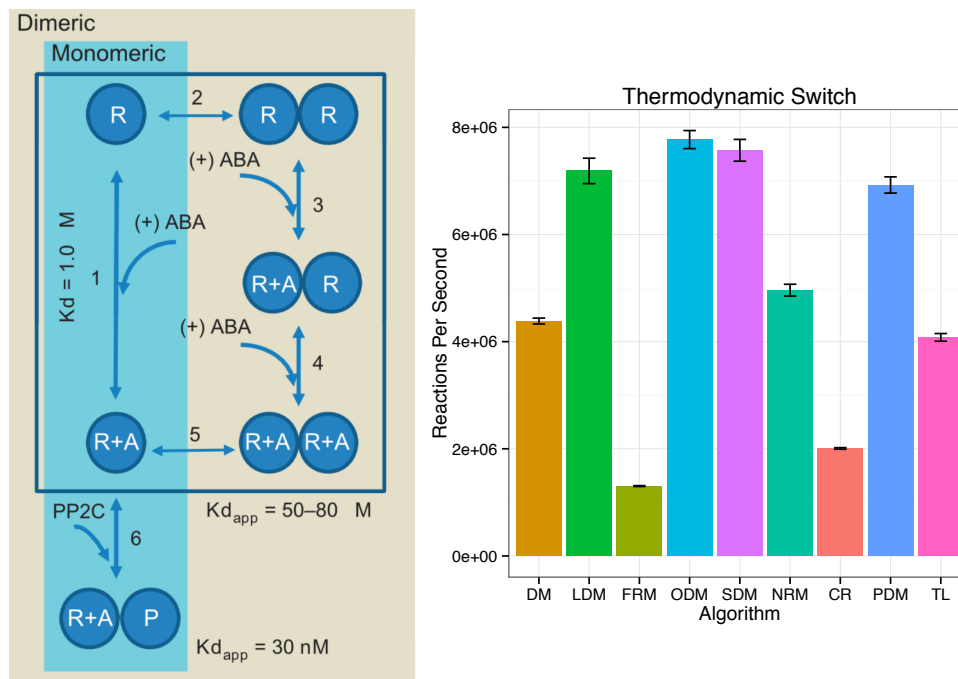


FIGURE 3.7: "Thermodynamic switch" abscisic acid (ABA) regulation model with SSA performance benchmark for nine different algorithms. The left hand side figure shows an overview of the abscisic acid (ABA) regulation model (taken from [13]). In this diagram, A represents ABA, R represents a receptor and P represents a PP2C phosphatase. The right hand side figure shows the benchmark results for this model over nine algorithms. The benchmark was performed on a single core of an Intel i7 2600K with 16GB RAM. The benchmark was run on each algorithm-model combination with error bars showing the variation over 10 samples.

The ABARs are divided between monomeric and dimeric oligomeric states. Dupeux et al. experimentally show that dimerisation prevents interactions between PP2Cs and ABARs unless ABA is present. They created a model (see left side of Figure 3.7) to test the competition of binding affinities between dimeric and monomeric receptor proteins. This model demonstrated that at low ABA concentrations monomeric have stronger binding affinities for receptor activation, whilst at higher concentrations both receptor types contribute equally to the process. These results lend weight to

their hypothesis that activation of signalling pathways is influenced by the thermodynamic effects of receptor oligomerisation [13].

The right side of Figure 3.7 shows the results of the SSA performance benchmark of the ABA regulation model (A4). ODM is the fastest algorithm for this model, closely followed by SDM and LDM (which are two quite similar algorithms to ODM). The SSA performance profile of model A2 is similar to model A4 with the exception of PDM. Whilst PDM is still a strong performing algorithm for model A4, it only ranks 4th overall (compared to first for model A2). Interestingly, this indicates that there are differences between these two models that only affects the relative performance of PDM compared to other algorithms. Overall algorithm perform is lower than for model A2, but this is to be expected as A4 is a larger model (see Table 3.1).

3.2.6 Model A5: Auxin transport case study

Auxin is an important plant hormone that influences growth and development, inducing cell elongation, division and differentiation [72]. The morphogenetic pattern formation effects of auxin do not simply occur as an outcome of reaction diffusion [1]; auxin is actively *pumped* through plant cells in a specific direction. When the concentration of auxin reaches a maxima at a plant tip, growth is induced. For example, if light is present at one side of a plant, auxin is pumped to the unlit side. Consequently, the unlit side of the plant is stimulated to grow at a faster rate than the lit side. This directional growth behaviour causes the plant to bend toward the direction of the light source in order to improve photosynthetic efficiency.

Auxin is transported directionally by *import* and *export* proteins. The *AUX1* protein acts as a cell auxin importer, whilst the *PIN1* protein behaves as the plant cell auxin exporter. *AUX1* recruits auxin from all positions surrounding the cell, but the *PIN1* exporter is positioned at a specific side of the cell wall to generate a directional flow of auxin (see Figure 3.8).

Twycross et al. introduced a compartmentalised multi-scale model of auxin transport designed to represent a row of contiguous cells segments in a plant stem [12]. This

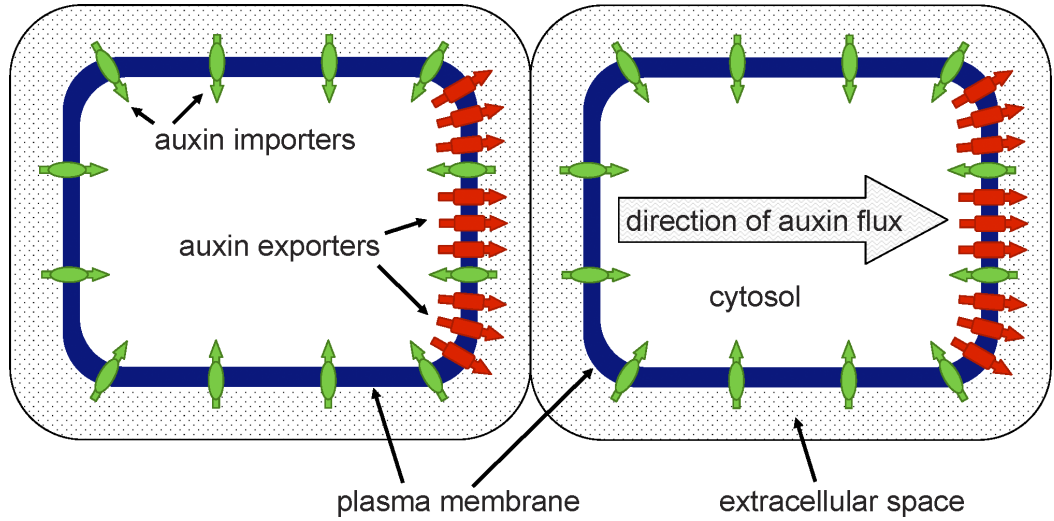


FIGURE 3.8: Auxin transport proteins in *Arabidopsis* cells (taken from [72]). Import proteins (green) recruit auxin from the extracellular space into the cytosol. Export proteins (red) transfer auxin from the cytosol into neighbouring extracellular space in a specific direction. The net effect is to create a pumped *polar* flow of auxin across plant cells.

model represents a standard experiment to measure auxin velocities using radio labelled agar to trace the flow of auxin molecules through the plant stem. The model is broken down into a row of stem segment compartments for spatial resolution, between source and sink agar compartments (see Figure 3.9). There are a total of 43 compartments in the model including 21 apoplast (extracellular space) compartments in alternating sequence with 20 cytoplasm compartments to model the plant stem. Both *in vitro* and *in silico* experiments measure the number of molecules that arrive at the sink block from the source block to determine the auxin *flux*.

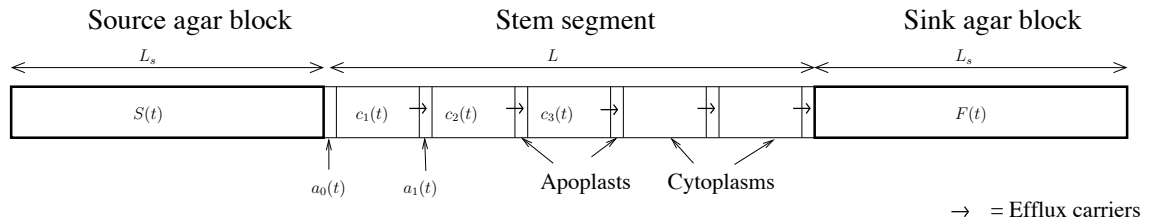


FIGURE 3.9: Auxin transport experiment model (taken from [12]).

This case study evaluates the accuracy of different modelling approaches: (1) discrete stochastic, (2) deterministic numerical solution and (3) deterministic analytical solution. Twycross et al. stress the importance of considering multiple modelling approaches when assessing a biological system, and pay particular attention to the

importance of stochastic modelling. They argue that stochastic modelling allows for a mechanistic understanding at the molecular level of this inherently multi-scale system, to observe tissue level phenomena caused by stochastic noise generated at the cellular level [12]. Wet lab experiments reveal the velocity of auxin to be approximately $1 \text{ cm} \cdot \text{h}^{-1}$. The deterministic asymptotic model determined an auxin velocity of $1.95 \text{ cm} \cdot \text{h}^{-1}$, whilst the stochastic simulation indicated $3.38 \text{ cm} \cdot \text{h}^{-1}$. The authors state that these are good predictions considering the resolution of the model and that some parameters are estimated. Furthermore, the sensitivity of the wet lab apparatus must be considered (minimum detection level) as the stochastic model reports the very first molecule entering the sink.

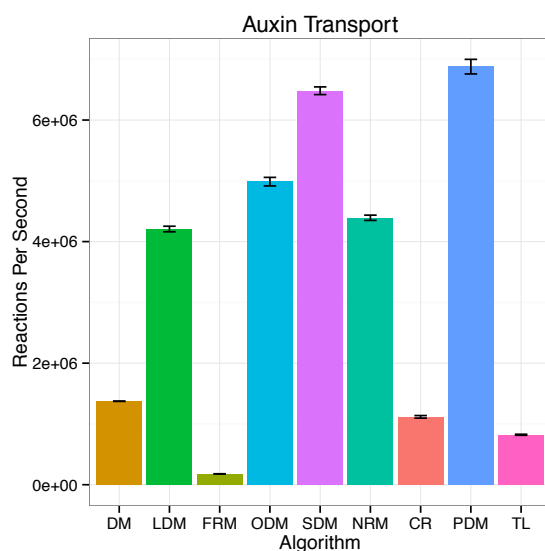


FIGURE 3.10: SSA performance benchmark for nine different algorithms of the Twycross et al. auxin transport model [12]. The benchmark was performed on a single core of an Intel i7 2600K with 16GB RAM. The benchmark was run on each algorithm-model combination with error bars showing the variation over 10 samples.

This auxin transport model is an example of a linear chain network topology [73] where reaction flow is “pipelined” through a single global chain of reaction channels. Figure 3.10 shows SSA performance for this model with PDM as the fastest algorithm for this network topology. Interestingly, TL has relatively poor performance even though the initial molecular species population for auxin is fairly high. This suggests that the linear chain network topology may bottleneck the performance of the TL algorithm. SDM is the second strongest algorithm for model A5 and has a higher

relative performance compared to ODM for any other model evaluated in the curated models dataset. This suggests that model A5 is subject to transient variations in reaction channel propensities.

3.2.7 Model A6: G Protein Signalling

Model A6 is the computational model produced by Heitzler et al. to unravel the signal transduction mechanisms controlling the angiotensin II type 1A receptor ($AT_{1A}R$) in human embryonic kidney cells [57]. The hormone angiotensin II is the strongest regulator of blood pressure in the human body, controlling vasodilation as well as water and salt balance [74]. $AT_{1A}R$ is a transmembrane protein, extending through the lipid bilayer of the cell membrane [75], weaving in and out of the membrane seven times. The $AT_{1A}R$ is a member of the of G protein-coupled receptor family which “activate” G proteins as part of a signal transduction pathway [74].

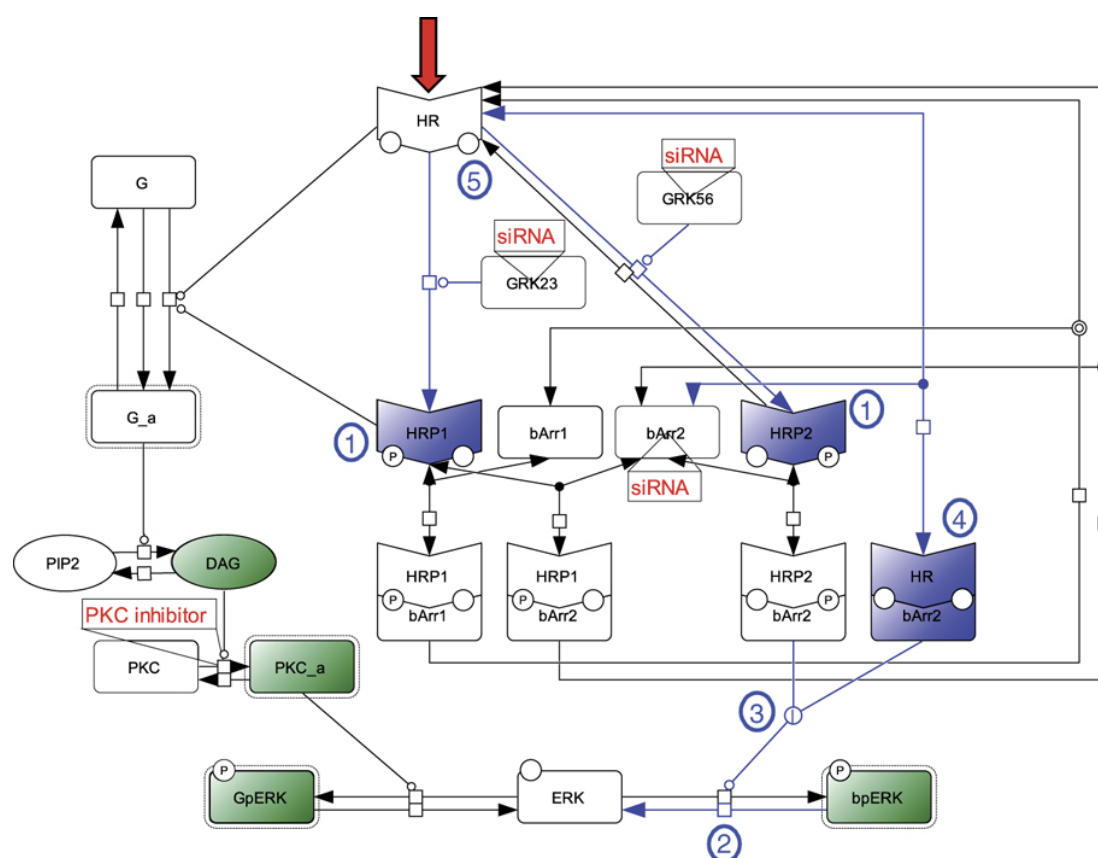


FIGURE 3.11: Overview of competing G protein-coupled receptor kinase signalling model (taken from [57]).

Figure 3.11 is an overview of the competing G protein-coupled receptor kinase signalling model. As knowledge of the signalling pathways is incomplete, the model was developed from multiple hypotheses and refined incrementally until results agreed with the experimental data for the system. This model attempts to elucidate the extracellular signal-regulated kinase (ERK) activation by $AT_{1A}R$ via the G protein and β -arrestin pathways. The G protein pathway is activated quickly but its action is temporal, whilst the β -arrestin pathway is slow to activate and has a sustained effect. Both pathways are regulated by competing G protein-coupled receptor kinases (GRKs) which are responsible for receptor phosphorylation.

The variable HR in the model encapsulates the entire hormone-receptor binding process whilst $HRP1$ and $HRP2$ are the receptor after phosphorylation by the competing GRKs. G proteins (G) are activated (G_a) under the catalytic effect of HR as well as the phosphorylated $HRP1$. However, the phosphorylated $HRP2$ state quenches G protein activation through depletion of HR but induces the β -arrestin pathway. G protein activation initiates a signalling cascade by catalysing the cleaving of the membrane lipid $PIP2$ into the second messenger DAG [76]. Second messengers are small intracellular signalling molecules that are produced upon receptor activation, rapidly broadcasting signals to other cell areas [75]. This in turn catalyses the activation of protein kinase C (PKC), which induces the phosphorylation of ERK through G protein-dependent mechanisms - noted as $GpERK$ in the model. The β -arrestin pathway catalyses β -arrestin-dependent ERK phosphorylation - noted as $bpERK$ in the model. Heitzler et al. consider both phosphorylated forms of ERK ($GpERK$ and $bpERK$) separately in the model as they have different physiological effects [57].

The competing G protein-coupled receptor kinase model considers a number of hypotheses (indicated as blue circled numbers in Figure 3.11), which were experimentally validated [57]. These are: (1) two distinct phosphorylated forms of receptor $HRP1$ and $HRP2$, (2) reversible β -arrestin-dependent ERK phosphorylation, (3) enzymatic amplification of β -arrestin-dependent ERK phosphorylation, (4) non-phosphorylated receptor can induce ERK activation and (5) two modes of receptor phosphorylation.

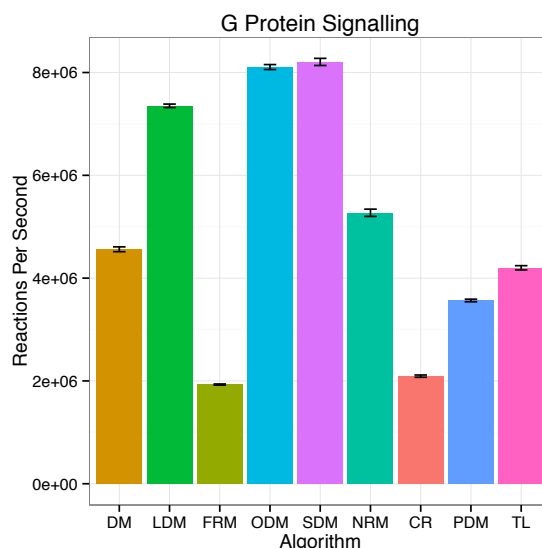


FIGURE 3.12: SSA performance benchmark for nine different algorithms of the Heitzler et al. competing G protein-coupled receptor kinase model [57]. The benchmark was performed on a single core of an Intel i7 2600K with 16GB RAM. The benchmark was run on each algorithm-model combination with error bars showing the variation over 10 samples.

As shown in Figure 3.12, SDM is the fastest performing algorithm for model A7. This implies that there are variations in reaction channel propensities over the course of the simulation, as SDM loosely adjusts reaction search depth based on reaction propensities. For the previous models evaluated, PDM performance has been greater than or only slightly less than that of SDM or ODM. However, in model A6 PDM performance is significantly lower than ODM or SDM. This reveals the existence of model configurations that are suboptimal for the PDM algorithm.

3.2.8 Model A7: Discrete proliferation model

Unlike the other models in the curated model dataset, model A7 is not specifically a biological model. It is a generic model that can be applied to many different fields including biology, ecology and finance [58]. The model was devised to demonstrate unexpected spatio-temporal behaviour that can emerge from discrete dynamic systems. This model encapsulates the idea of discrete agents that proliferate and die, echoing the way that biological cells divide and perish. Another important feature

of this model is that it is two dimensional and can be imagined as populations on a grid. Parallels can be drawn between this system and the famous “Conway’s game of life” cellular automaton [77].

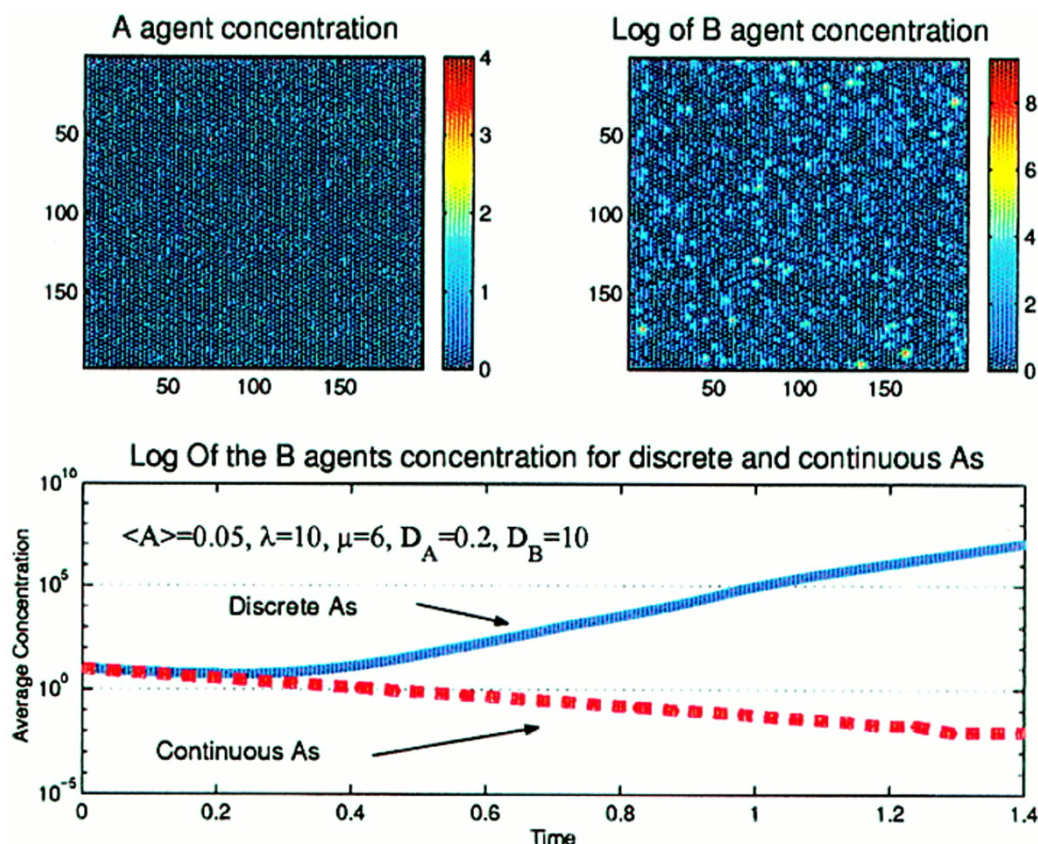


FIGURE 3.13: Results of the Shnerb et al. discrete proliferation model for both continuous and discrete agent simulations (taken from [58]). The top left figure shows a snapshot of the concentrations of type *A* agents in the two-dimensional model. The top right figure shows a snapshot of the concentrations (on a logarithmic scale) of type *B* agents. The bottom figure shows the time-series of *B* agent populations for a discrete simulation (solid blue line) and continuous approximation (dashed red line).

More specifically, my SSA simulatable realisation of the Shnerb et al. model is a 10 by 10 two-dimensional lattice upon which two types of agent (*A* and *B*) can exist at each lattice point. In this model implementation, agents are represented as “molecular species” which follow typical SSA “reaction” rules. Agents of type *A* are *immortal* (i.e. are not degraded, consumed or subject to death), whilst agents of type *B* *die* with a probabilistic rate μ . Both agent types can move around the lattice positions; rules which are realised as diffusion reactions with probabilistic rates D_A and D_B

respectively. If the two agent types meet on a lattice point during the simulation, B agents can divide at a rate of λ . This rule implies that type A agents act as catalysts to B agent proliferation. With the assumption that the proliferation rate λ of B is lower than the death rate μ , a continuous approach would predict that the B population would decrease and eventually become extinct. Equation 3.1 shows B time variation as a continuous partial differential equation model [58].

$$\frac{\partial n_B}{\partial t} = D_B \nabla^2 n_B + (\lambda n_A - \mu) n_B. \quad (3.1)$$

However, an “exact” discrete approach reveals that individuals self organise into spatio-temporal groups to survive and prosper [58]. This is experimentally validated by the timeseries plot of Figure 3.13 with the continuous approximation (red dashed line) showing the B population to exponentially decrease, though the discrete method (solid blue line) declares exponential B population growth. It is generally assumed that continuous approximations are suitable for systems considered at the macroscopic scale, such as this lattice scale growth model. Shnerb et al. demonstrate that discrete, stochastic fluctuations present at the microscopic scale can have a pronounced effect at the macroscopic scale. The top right sub-figure of Figure 3.13 shows that the populations of B agents self organise into patchy structures at the macroscopic level.

Figure 3.14 presents the results of the SSA performance benchmark of my lattice implementation of the Shnerb et al. proliferation model. Species amounts are initially low at $t = 0$, but the exponential growth of the B species population leads to drastic changes in SSA performance profiles as the simulation progresses. At $t = 0$, TL is the slowest algorithm and has extremely poor performance compared to the fastest algorithm PDM. Conversely, by $t = 200$ TL performs orders of magnitude faster than any other SSA formulation. The reaction network is large compared to other models in this dataset at 920 reactions (see Table 3.1). There are 200 species in the model but these are actually the two agent types occupying the 100 lattice points. As so many reaction channels are funnelled through relatively few species, the network is

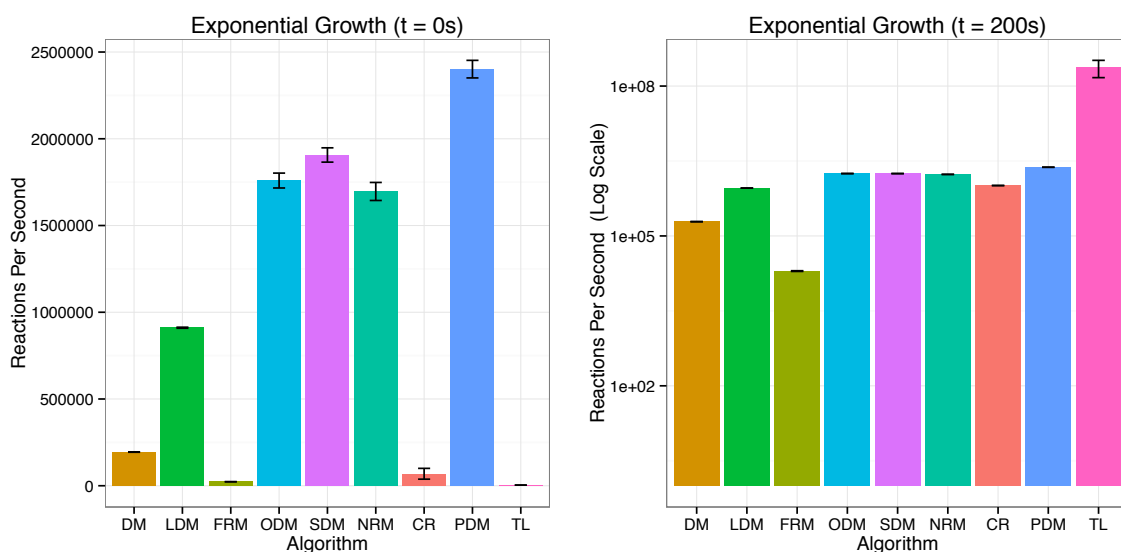


FIGURE 3.14: SSA performance benchmark of the Shnerb et al. discrete proliferation model for nine different algorithms. The left hand side figure shows the benchmark results when run at $t = 0$ simulation time. The right hand side figure shows the benchmark results when run at $t = 200$ simulation time. The benchmark was performed on a single core of an Intel i7 2600K with 16GB RAM. The benchmark was run on each algorithm-model combination for 10 seconds of CPU time with error bars showing the variation over 10 samples.

tightly coupled which explains strong PDM performance. This benchmark provides clear evidence that the transient nature of stochastic simulations impact upon simulation performance. Whilst it would be preferable to pick the fastest algorithm for a given model, the system may reach states which indicates that a dynamic change of algorithm would be optimal.

3.2.9 Model A8: Stochastic model of *lacZ lacY* gene expression

Model A8 investigates stochastic behaviour in simple prokaryotic gene expression. Kierzek et al. introduced a model of single gene expression during the exponential growth phase of a cell [78]. Based on experimental data, this system specifically modelled *lacZ* gene expression in *Escherichia coli*. The *lacZ* gene is part of the lac operon (see left side of Figure 3.15) and codes for the β -galactosidase enzymatic protein. In *Escherichia coli* the lac operon enables the metabolism of lactose.

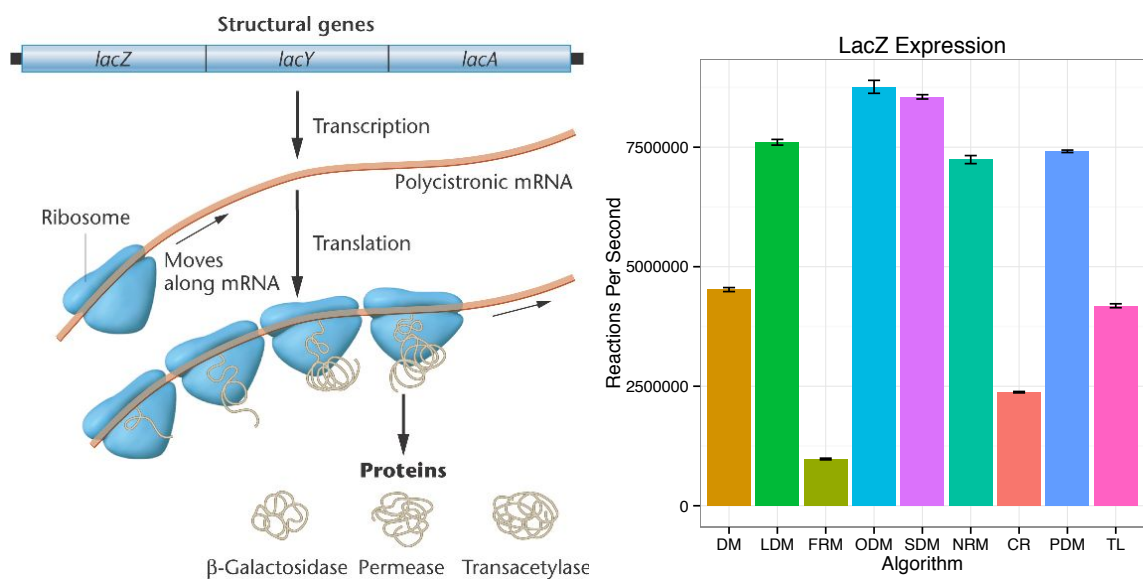


FIGURE 3.15: Model of *lacZ lacY* gene expression with SSA performance benchmark for nine different algorithms. The left hand side figure shows the transcription and translation of the *lac* operon (taken from [79]). The right hand side figure shows the benchmark results for the *lacZ lacY* gene expression model over nine algorithms. The benchmark was performed on a single core of an Intel i7 2600K with 16GB RAM. The benchmark was run on each algorithm-model combination with error bars showing the variation over 10 samples.

This model goes into fine grained detail and considers the biomechanics of transcription and translation timings. The model results agree generally with experimental data [78] and reveals that stochastic transcription behaviour is dependent on promoter strength. Promoter strength is tuned by adjusting the binding rate of RNAP with the *lac* operon promoter. Using this model, Kierzek et al. found that strong promoters tend to produce uniform gene expression whilst weak promoters produce bursts of transcription associated with stochastic behaviour. Understanding the effects of promoter strength is important for synthetic biology applications to accurately control the levels of gene expression [80].

Kierzek further extended the *lacZ* model to include *lacY* expression, creating the *lacZ – lacY* model that I have benchmarked in this section (see Figure 3.15) [78]. *lacY* encodes the lactose permease protein which has the role of transporting lactose directionally into the cell. In this model, *lacZ* and *lacY* are expressed constitutively which means they are constantly active. This is because the model was designed to

test computational limits [59] and represents an *Escherichia coli* mutant that lacks a lac repressor.

Kierzek et al. state that transcription, translation and mRNA degradation are “tightly coupled” [59]. For example, their model makes the assumption that ribonuclease and ribosomes compete for the RBS which implies reaction network coupling. A scientist might expect PDM to be the optimal algorithm given this knowledge of the model and claims by the authors of the PDM algorithm [29]. However, the SSA performance benchmark for this model (see right side of Figure 3.15) reveals that ODM is the best performing algorithm for this model, whilst PDM only ranks 4th.

3.3 Summary & conclusions

This chapter has introduced the requirements for stochastic simulation of biochemical models by presenting the process of modelling synthetic Boolean gates. A modeller begins with a hypothetical mechanistic outline of the biosystem (see Figure 2.4) which is then distilled into a set of stochastic rules between molecular species (see Table 2.4). With these rules and species, along with specified reaction rate parameters and initial species amount information, a stochastic simulation can be executed. The output for the SSA is a timeseries log of the (species) state vector which can then be analysed to evaluate the model hypothesis. My benchmarks of the synthetic Boolean gate models demonstrates that it is possible to see large variations in SSA performance caused by differences in initial species amounts.

Section 3.2 concisely describes models taken from systems and synthetic biology literature that were subsequently benchmarked for my initial treatise of SSA performance. A total of nine model benchmarks were performed as model A7 was benchmarked for two sets of initial species amount values. One observes that TL and PDM were the fastest algorithms for three models each, whilst ODM and SDM are the fastest for two models and one model respectively. Naturally, this means that five of the nine SSA variants evaluated did not achieve a top ranking status for any of the nine models. These include two formulations that I have found to be a popular choice for

stochastic simulation. Firstly, DM (which can be considered the *de facto* standard SSA formulation) is trivial to comprehend and implement when compared to more modern SSAs. This simplicity makes DM an attractive choice for a scientist or developer who needs to integrate stochastic chemical kinetics into their simulation software. However, DM was significantly slower than the fastest algorithm for each benchmark. Secondly, NRM, which features multiple optimisations is available within established simulation software. NRM was the first major advance in SSA technology since DM, and has had more time to percolate through the computational biology community than the most recent variants. Whilst NRM had a higher average ranking than DM over the models investigated, based on these results one could not advise that it is simply applied to all simulations if performance requires consideration.

If one compares the benchmarks on a model by model basis, one can observe three distinct “classes” of model-algorithm performance profile. Models A1 and A7($t = 200s$) belong to a class of model where TL greatly outperforms any other algorithm. Models A7($t = 0s$) and A5 benchmark results belong to the second class of model-algorithm results and have the same algorithm rankings. PDM was the fastest algorithm for this second class of model. The final class consists of five models: A2, A3, A4, A6, and A8. Whilst there are obvious similarities in performance profiles, this class has larger variability in actual algorithm rankings. For example, PDM is the fastest for A2 but only ranked fourth for model A4, but the relative performance differences are small. The biggest variation is present in model A6 which has a large drop in performance for PDM compared to the other models. If I sum the rankings of the strongest algorithms for this model class I find that ODM scored 7, SDM scored 10 and PDM scored 19. This means that one can consider ODM as the most consistent algorithm for this class of model.

From these benchmark experiments on published models, one cannot declare any one SSA formulation to be superior to all others. In fact, one can observe that several SSAs are capable of ranking first depending on the specific model simulated. This lends weight to the first hypothesis of this thesis: *There is no single SSA that is superior in performance for every biomodel*. Furthermore, I have shown some preliminary

findings that model-algorithm performance can be clustered into classes of similar SSA performance profiles. Evidence has been presented that algorithm performance is dependent on the specific model and that different types of model share similarities in overall SSA performance profiles. A connection between model class and algorithm performance implies that model characteristics influence performance. This evidence supports the second hypothesis of the thesis: *There is a relationship between biomodel characteristics and SSA performance.*

One can observe from model A7 and the synthetic Boolean gates benchmarks that differing initial species amounts can also cause variations in algorithm performance. Chapter 5 demonstrates my methodology to determine model characteristics using static topological properties (thus ignoring species amounts). In Chapter 6 I will demonstrate that is indeed possible to make accurate predictions of SSA performance using only the static topological properties.

Chapter 4

Characterising Biochemical Models

4.1 Introduction

This chapter investigates the quantitative characterisation of stochastic biochemical models. Two datasets of biochemical models are introduced that will enable the performance and predictions experiments contained in this thesis. Biochemical models can be represented as graphs from which one can extract topological data to find relationships (and differences) between them. The stochastic simulation literature only considers a few model properties that it is assumed are related to algorithm performance such as reaction network size and the level of coupling between reactions. I have performed an exhaustive appraisal of graph properties for the biochemical model datasets which I will use in later chapters to test the hypotheses of this thesis.

“ Most *[biological]* network papers discuss at most two or three metrics at a time. What justifies the choice of a few metrics, in place of a comprehensive suite of network metrics? Is there any scientific basis of the choice of the metrics or are they invariably handpicked? More importantly, do these few handpicked metrics carry the maximum information extractable about the biological system?

”

Soumen Roy [81]

4.2 Computing properties of biochemical models

4.2.1 Biochemical models as graphs

Understanding biological systems requires scientists to abstract the source of their complexity. One can catalogue the components of a biosystem, but one must also inspect the immense web of internal interactions that allow it to maintain homeostasis or to change state.

Kitano states that systems biology observes four key points in order to gain a “systems” level understanding of a complex system: (1) System structure, (2) System dynamics, (3) Control mechanism, and (4) Design methods [2]. Simulation allows scientists to investigate the dynamics of a system and elucidate control mechanisms. *Structure* must be modelled by abstracting the *network* of interactions in the system, so that common design patterns or *motifs* can be found from the observation of dynamics in the context of network topology.

“*Network biology*” uses *mathematical graphs* to represent cellular networks [82]. Network analysis sits at the foundations of systems biology aided by computational modelling and analytical tools [83, 84]. In this thesis, my focus is on the transcriptional regulatory networks that can be modelled as *directed graphs* (see Section 4.2.3). Complex biological systems expressed as mathematical graphs can be analysed with well established graph theoretical methods [82]. Graph topology can predict the complex behaviours that govern biosystems and thus one would intuitively expect to find a relationship between topology, system dynamics and ultimately simulation performance.

4.2.2 Experimental models

In the previous chapter, I used a small “cherry picked” set of curated fully parametrised models (see Section 3.2.1). A comprehensive benchmark of SSA performance requires an extensive dataset for a statistically meaningful analysis of algorithm behaviour. Thus, I now introduce a second dataset containing 380 models in SBML [33] format retrieved from the *BioModels database* [85]. In Figure 4.1, a histogram is shown displaying the spread of model size within the dataset, quantified by *reaction_num_vertices* (which equates to the reaction network size of a model). It can be seen from the histogram that the vast majority of BioModels have a reaction network size of 50 reactions or less, but there are a small number of larger models (up to 1800 reactions) also used for the analysis.

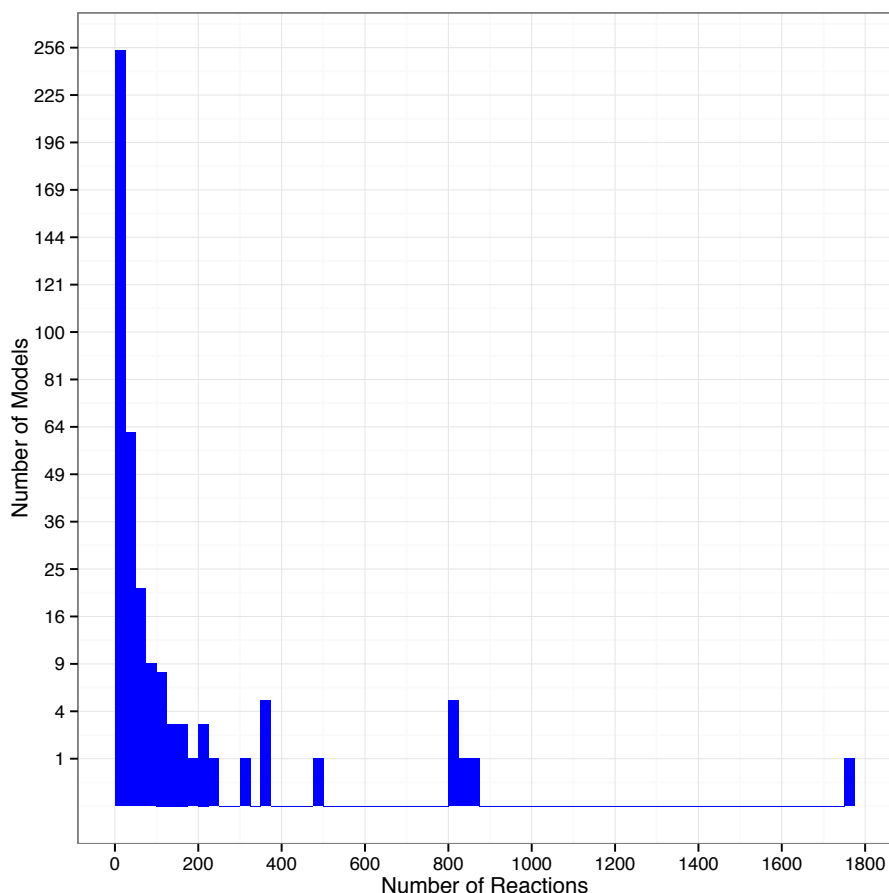


FIGURE 4.1: Histogram displaying spread of model reaction size within the BioModels dataset. Number of reactions equates to the reaction size of a model. The x-axis bin size is 25. The y-axis is on a square root scale.

The BioModels usually contain deterministic rate functions instead of stochastic rate constants, and thus a decision was made to set the stochastic rate constants of all reactions to 1.0. This essentially converts them into non-deterministic models. This decision also means that property analysis is performed upon *unweighted* dependency graphs derived from these models (i.e. graphs lacking reaction rate data). Furthermore, in order to simplify models and remove extra variables that cannot be captured by the static dependency graph analysis, the amounts of all species were set to 100 and remain constant throughout simulation.

It should be noted that this means that any analysis performed on the BioModels will not be able to account for transient changes within a simulated model. The curated models dataset (see Section 3.2.1) is completely parametrised but small in size (8 models) and not sufficient for definitive analysis. Therefore, analytic insight garnered from the BioModels dataset will be tested using the curated models dataset. I wish to highlight that whilst there are many complete deterministic models available from online databases, few complete curated stochastic models are freely available. Therefore, a future analysis featuring complete stochastic models will have to be preceded by the creation of a dataset with a reasonably large number of curated stochastic models. This is an open challenge for the systems and synthetic biology communities at large.

4.2.3 Graphs and graph theory

Graph theory is a branch of discrete mathematics that abstracts the representation of discrete entities and their relationships. Graphs consist of symbolic points (referred to as nodes or *vertices*) that are connected by symbolic lines (referred to as *edges*). Each vertex will typically represent an entity and an edge will represent a relationship between a pair of vertices. These relationships can be bi-directional (*undirected* graph) or uni-directional (*directed* graph), with directed edges possessing an arrow head to signify relationship directionality. Meta-data can be associated with edges or vertices by graphs to *label* the graph. Graphs can also be *weighted* which means

they have associated edge value labels, for example to represent distances if vertices were equivalent to geographical locations.

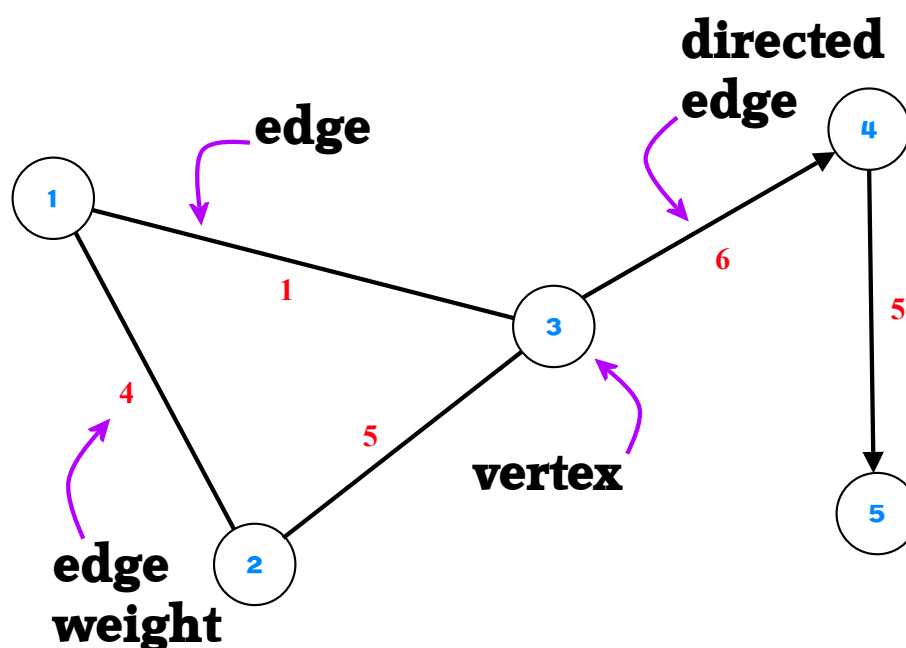


FIGURE 4.2: Diagram of an example graph. This graph is *weighted* and each of the 5 vertices is labelled. The graph contains 5 weighted edges, 2 of which are *directed*.

The graph paradigm can be applied to a diverse array of discrete subjects, for example algorithms can be represented as graphs and a subsequent analysis can measure the algorithm's computational complexity. Networks are analogous to graphs, for example in a social network each vertex represents a person and an edge represents a nominal friendship between 2 people (i.e. 2 vertices). Analysis of the social network graph can reveal relationships such as friendship groups and predict which friends have not yet made connections.

Biochemical models consist of molecular species entities and a set of reactions that define the relationship between species. Therefore, there is a concomitant mapping from a biochemical model to a species network graph (SNG) (see Section 4.2.5). A reaction dependency graph (RDG) can also be generated by setting reactions as vertices and edges to represent the species dependency relationship between reactions (see Section 4.2.4).

4.2.4 Exemplar reaction network & reaction dependency graph (RDG) generation

In this section, I introduce an example reaction network (i.e model) upon which I demonstrate the generation of a RDG. Table 4.1 lists the reactions in an example reaction network and the resulting dependencies for each reaction. I define dependencies as the reactions that need to be updated when a reaction is executed at each iteration of the SSA.

Name	Reaction	Depends	Affects	Update
R1	$A \rightarrow B$	A	A, B	R1, R2
R2	$B \rightarrow C$	B	B, C	R2, R3
R3	$C + D \rightarrow E$	C, D	C, D, E	R3, R4, R6
R4	$E \rightarrow E + F$	E	F	R5
R5	$F \rightarrow A$	F	A, F	R1, R5
R6	$E \rightarrow B$	E	B, E	R2, R4, R6

TABLE 4.1: Example reaction network and reaction dependencies from *McCollum et al* [27].

It is important to note the *affects* column of Table 4.1 as this operation allows reaction dependencies to be calculated for each reaction. The species that are affected by a reaction are those whose values are changed when the reaction is executed. *R4* ($E \rightarrow E + F$) illustrates this by only affecting *F*, because the net effect on the population of *E* is zero when the reaction is executed. Whilst a reaction of this type is chemically implausible, it can be used to represent a gene transcribing a protein in a high level biochemical model ($Gene \rightarrow Gene + Protein$).

After a reaction is executed, if the affected species are members of the set of species any reaction *depends* (see Table 4.1) upon, then those reactions need to be updated. For example, executing *R1* affects species *A* and *B*. Therefore, *R1* needs to be updated as it depends on *A*. *R2* needs to be updated as it depends on *B*.

The naive method of generating a RDG is shown in Algorithm 15. This method works by iterating through each reaction and testing whether it *affects* any of the other reactions if executed, resulting in $O(M^2)$ scaling. The resulting RDG based on the reaction network in Table 4.1 is visualised in Figure 4.3.

Reaction Dependency Graph (RDG)

In a reaction dependency graph, each vertex corresponds to a unique reaction, hence the number of vertices in a reaction dependency graph is equal to the number of reactions in the model. A directed edge is placed from vertex V_i to vertex V_j if the firing of reaction R_i changes the propensity of reaction R_j . Any duplicate edges are removed from the graph.

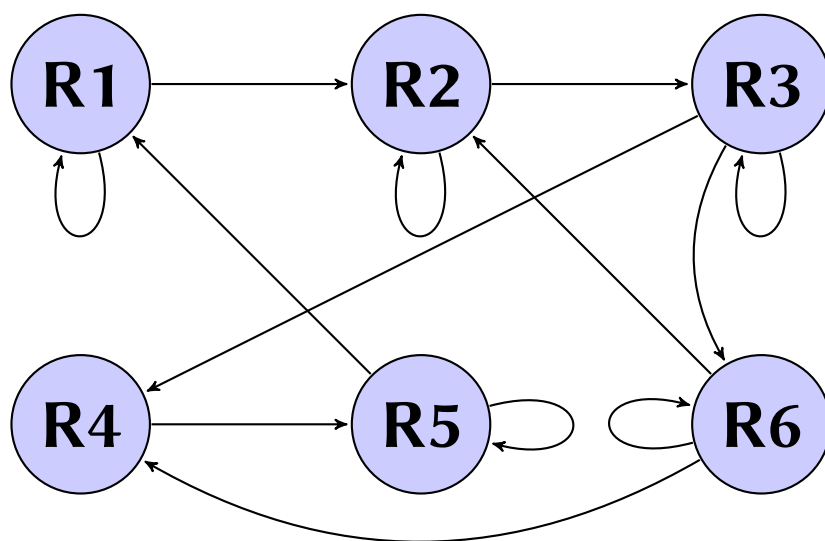


FIGURE 4.3: Reaction Dependency Graph (RDG) generated from exemplar reaction network in Table 4.1. In this graph, directed edges point to the reactions which need to be updated (propensity recalculated) when a particular reaction is executed.

4.2.5 Species network graph (SNG) generation

The species network graph of Table 4.1 is shown in Figure 4.4. This type of graph maps to the reaction network with negligible processing or transformation. Put simply, each species is a vertex in the graph and a directed edge is placed between each product and reactant of every reaction in the model.

Species Network Graph (SNG)

In a species network graph, each vertex corresponds to a unique species, and so the number of vertices in a species network graph is equal to the number of species in the model. A directed edge is drawn from vertex V_i to vertex V_j if for any reaction species S_i is a reactant and species S_j is a product. Any duplicate edges are removed from the graph.

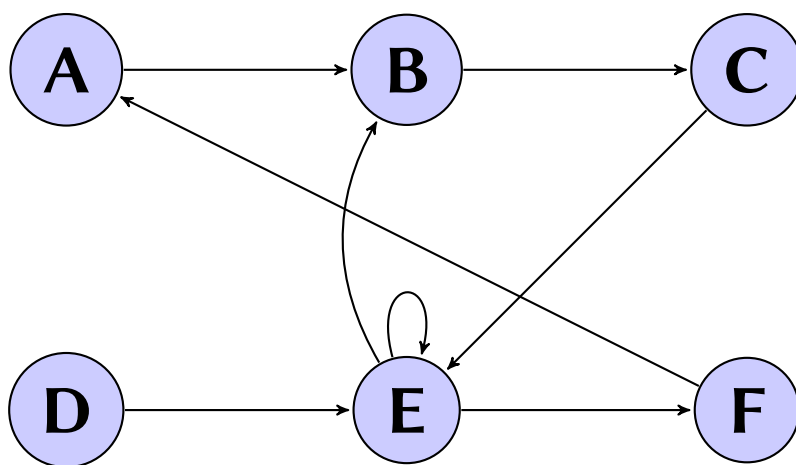


FIGURE 4.4: Species Network Graph (SNG) generated from exemplar reaction network in Table 4.1.

This graph results in a visualisation that a biologist would typically create to understand a biosystem. However, the layout would usually be less condensed than in Figure 4.4, with edge overlaps avoided if possible to improve human comprehension of the system.

Even in the small example model from Table 4.1, the connectivity profile of the SNG and RDG are quite distinct. This indicates that there are different features available from analysing both model-graph interpretations individually.

4.3 Analysis of graphs

Models were characterised by calculating the values of a wide range of graph properties of the *reaction dependency graph* and *species network graph* of every model. The reaction and species graph properties which were calculated using the *igraph* C library [86], are summarised in Table 4.2. Properties which relate to individual or subsets of vertices or edges were calculated over all vertices or edges in the graph, and the minimum, maximum and mean values recorded. Where it was possible to calculate a property considering the graph as undirected (i.e. replacing all directed edges by undirected edges), or only using incoming or outgoing edges, then values for both directed and undirected, or incoming and outgoing edges were also calculated.

4.3.1 Number of graph vertices and edges

The simplest graph properties to quantify are the number of vertices V and edges E . These values can typically be found by querying the size of the data structures that describe the graph (hence $O(1)$ time complexity). In the RDG, the number of vertices is equivalent to the number of reactions in the graph, whilst vertices equate to species in the SNG.

$$V = \text{Number of Vertices} \quad (4.1a)$$

$$E = \text{Number of Edges} \quad (4.1b)$$

Computational Complexity	Graph Property
$O(1)^\dagger$	number of edges, number of vertices, density of graph
$O(V)^\dagger$	min mean max outgoing edges, min mean max incoming edges, min mean max all edges
$O(V + E)^\dagger$	weakly connected components, articulation points, bi-connected components, reciprocity of directed graph
$O(VE)$	average geodesic length (undirected), average geodesic length (directed), min mean max outgoing closeness, min mean max incoming closeness, min mean max closeness in undirected graph, min mean max betweenness, min mean max betweenness in undirected graph, min mean max edge betweenness, min mean max edge betweenness undirected graph
$O(V(V + E))$	min mean max shortest path in undirected graph, min mean max shortest incoming path, min mean max shortest outgoing path
$O((V + E)^2)$	girth of undirected graph
$O(Vd^2)$	transitivity of graph vertices, average local transitivity
$O(V^4)$	min edge connectivity
$O(V^5)$	min vertex connectivity

TABLE 4.2: Summary of model topological properties analysed. Complexity relates to *worst case time complexity* for the computation of the property, where V is vertices, E is edges, and d is the average node degree. Properties marked with † have constant or linear scaling and are known as the restricted set of *fast* properties.

4.3.2 Graph density

Graph *density* is a measure of how densely interconnected a graph is with regard to its edges. This is equivalent to the term *degree of coupling* used by biologists when referring to a reaction network model. Graph density is a computationally trivial property to calculate ($O(1)$) as the expression to compute it only relies on V and E (see Equation 4.2).

$$density = \frac{2|E|}{|V|(|V| - 1)} \quad (4.2)$$

Equation 4.2 assumes that the graph is undirected, thus one ignores edge directionality of the RDG and SNG when calculating this property.

4.3.3 Graph degree

The *degree* of a vertex, is a count of the number of edges that are connected (*incident*) to it. For the special case of loops (i.e. an edge from one vertex to itself), the edge is counted twice. When considering a directed graph, one can restrict the counting to *incoming* or *outgoing* edges. To condense graph degree analysis, I record the *min*, *mean* and *max* vertex degree values for each of the incoming, outgoing and total edges. Computational complexity of degree calculations is $O(V)$, because one needs to iterate through each vertex in the graph and count the number of edges incident to it.

4.3.4 Weakly connected components

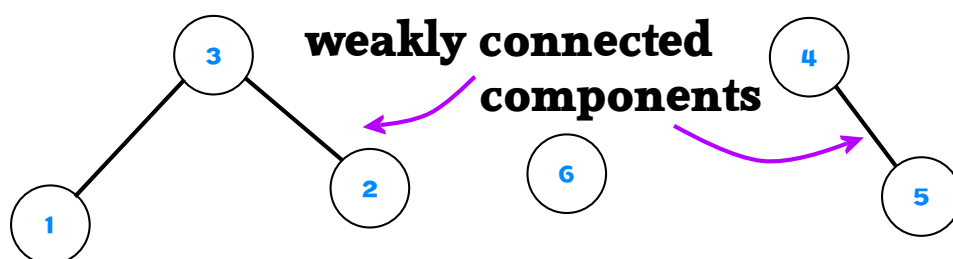


FIGURE 4.5: Diagram of a graph made up of 6 vertices that possesses 2 weakly connected components. Each vertex set $\{1, 2, 3\}$ and $\{4, 5\}$ are weakly connected components. Each weakly connected component holds the *maximal* connected property, as if (for example), vertex 6 was added to subgraph $\{4, 5\}$ forming subgraph $\{4, 5, 6\}$, the subgraph would no longer be connected.

A *connected graph* is one where there is a path between any two vertices in the graph. A *connected component* of an undirected graph is a subgraph of a supergraph that contains a path between vertices (i.e. the subgraph is connected), but is not connected

to other vertices in the supergraph. A subgraph is *maximal* connected if connecting any more vertices from the supergraph results in the subgraph no longer retaining connected status. *Weakly connected graphs* are directed graphs (such as the RDG and SNG) that are *maximal* connected when edge directionality is ignored. Thus, *weakly connected components* are maximal connected subgraphs of a supergraph. This can be calculated using a “backtracking” depth first search method, in $O(V + E)$ time[87].

4.3.5 Articulation points

Articulation points are vertices in a connected graph, that if removed result in the graph no longer being connected. Thus, removal of an articulation point increases the number of connected components in a graph. Tarjan and Hopcroft’s method can be used to find the articulation points in a graph [87]. The algorithm is based on a single pass of depth first search, hence has linear $O(V + E)$ time complexity.

4.3.6 Biconnected components

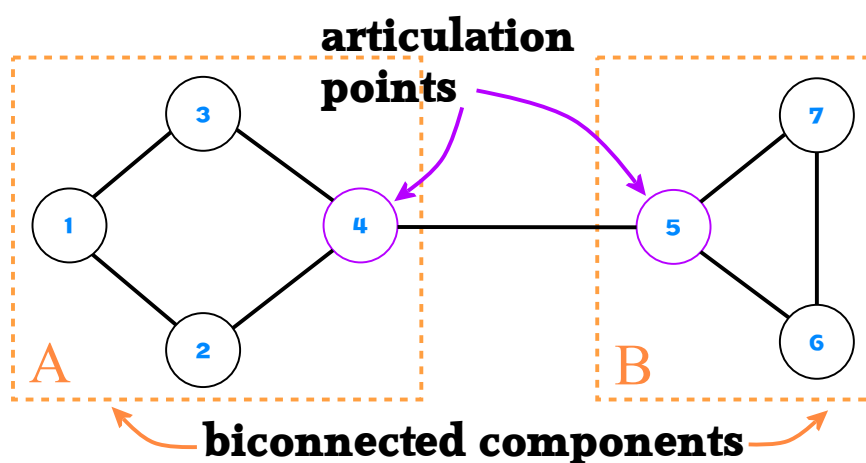


FIGURE 4.6: Diagram of a graph possessing 2 articulation points and 2 biconnected components. Vertex sets $\{1, 2, 3, 4\}$ and $\{5, 6, 7\}$ are from biconnected components marked *A* and *B* respectively. Vertices 4 and 5 are *articulation points* that separate the 2 biconnected components *A* and *B*.

A biconnected graph is a graph that contains no articulation points. Biconnected components of a graph are maximal biconnected subgraphs which are connected to

other biconnected components by articulation points. Biconnected components can be found using Tarjan and Hopcroft's method in linear $O(V + E)$ time [87]. This property is an indicator of network redundancy and thus robustness.

4.3.7 Directed graph reciprocity

Reciprocity of a directed graph is a measure of edge bidirectionality in a graph. The model analysis in Section 4.4 uses the ratio of the number of bidirectional connections L_B and the total number of connections L (see Equation 4.3). This computation of edge directionality can be made in linear $O(V + E)$ time.

$$\text{reciprocity} = \frac{L_B}{L} \quad (4.3)$$

4.3.8 Shortest paths in graph

For this measure, the shortest path from each vertex v in the graph to every other vertex is observed. One could consider this a measure of the size of the graph in terms of the “spread” of connected vertices. This measure can be contrasted to the size of the graph in terms of simply counting the number of vertices. The shortest paths for all vertex pairs in the undirected and directed graph are found, then minimum, mean and maximum values are recorded.

In an unweighted graph, $O(V + E)$ breadth first search can be used to calculate shortest path for an arbitrary vertex. Thus to calculate all shortest paths for all vertices the computational cost is $O(V(V + E))$. As a side note, Dijkstra's algorithm would be used in order to calculate shortest paths for a fully weighted graph and the Bellman-Ford algorithm for a partially weighted (or negatively weighted) graph.

4.3.9 Centrality

Centrality is a measure of how “central” a particular vertex is in a graph, or in other words, how much of a “focal point” a vertex is within a graph [88, 89]. Section 4.3.3 discusses the use of degree as a network property which one should note is a measure of centrality known as *degree centrality*. Degree centrality measures the number of incident edges for a vertex, and is thus a *local* centrality measure as it only considers connectivity to vertices that are adjacent to the vertex of interest.

In Section 4.4, I analyse 2 further measures of centrality: *closeness* and *betweenness*. Closeness and betweenness centrality differ from degree centrality in that they consider non-adjacent vertices and can be regarded as measures of *global* centrality. The closeness and betweenness centrality measures are defined in Sections 4.3.9.1 & 4.3.9.2.

4.3.9.1 Closeness centrality

Closeness uses (shortest path) distance as a metric to quantify the level of vertex centrality within a graph. Closeness centrality of a vertex v is defined as the inverse sum of the shortest paths between v and all other vertices (see Equation 4.5) [88]. Equation 4.4 defines $d(v, t)$ as the shortest path between v and an arbitrary target t (where h are intermediate vertices).

$$d(v, t) = \min(x_{vh} + \dots + x_{ht}) \quad (4.4)$$

$$\text{closeness}(v) = \left[\sum_{t=1}^N d(v, t) \right]^{-1} \quad (4.5)$$

However, this method is not appropriate for graphs with disconnected components [88], as there is an infinite distance between disconnected vertices. Opsahl shows that this can be remedied by rearranging Equation 4.5 to be the sum of inversed

distances (rather than the inverse of the sum of distances). This rearrangement is valid because the limit of 1 when divided by infinity is zero (see Equation 4.6).

$$\text{closeness}(v) = \sum_{t=1}^N \frac{1}{d(v, t)} \quad (4.6)$$

Closeness centrality can be computed using Newman’s method [90] in $O(VE)$ time. Newman’s method employs Dijkstra’s algorithm to calculate shortest paths [91].

4.3.9.2 Betweenness centrality

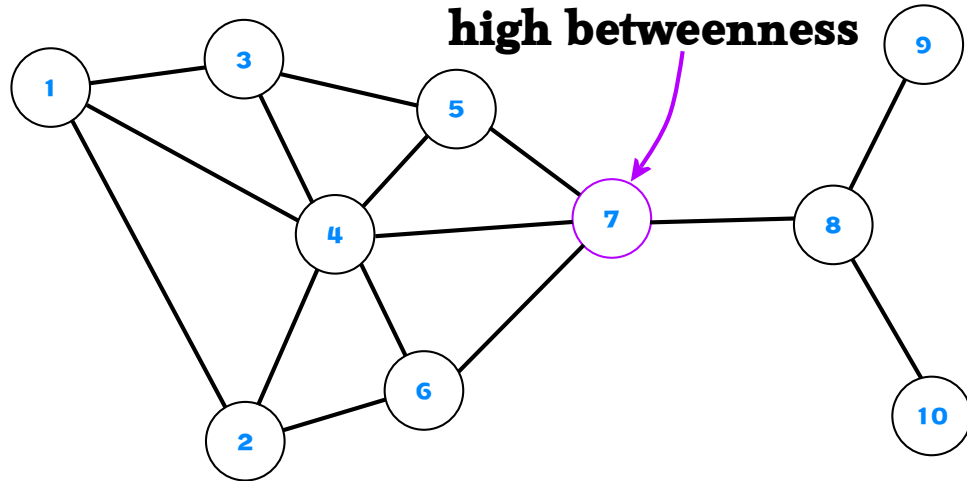


FIGURE 4.7: Diagram of a graph possessing a high betweenness vertex. Vertex 7 has high betweenness, and if removed would catastrophically damage the network.

Betweenness measures the likelihood a vertex v is found to be an intermediate vertex on the shortest path of 2 arbitrary vertices s, t of a graph (see Equation 4.7). This means that a vertex with high betweenness is in a position of “control”, as an important intermediate to relay information across a network [92]. Therefore, betweenness is also a measure of system robustness, as the removal of a high betweenness node may have a devastating effect on the network [93]. As an example, removal of vertex 4 in Figure 4.7 (a vertex with high degree centrality) would not seriously damage the network, but removal of vertex 7 (high betweenness centrality), would damage the network severely. In terms of algorithm performance, this may

indicate a potential bottleneck as the flow of network behaviour may be funnelled through this vertex [88].

$$\text{betweenness}(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (4.7)$$

Betweenness centrality can be computed using Brandes' method [94] in $O(VE)$ time. Brandes' method employs Dijkstra's algorithm to calculate shortest paths [91]. Edge betweenness is analogous to betweenness, but considers edges rather than vertices.

4.3.10 Average geodesic length

A geodesic in graph theory is the shortest path between 2 vertices. This particular metric measures all the shortest paths in the graph and finds the average. The average geodesic length can be computed in $O(VE)$ using Dijkstra's algorithm [91].

NOTE: The average geodesic length metric I measure is closely related to the mean shortest path metric. Whilst geodesic length is synonymous with shortest path, the shortest path metrics gather paths for all vertices including the edge case of V_i to V_i . However, the average geodesic length metric excludes the path from V_i to V_i .

4.3.11 Girth of graph

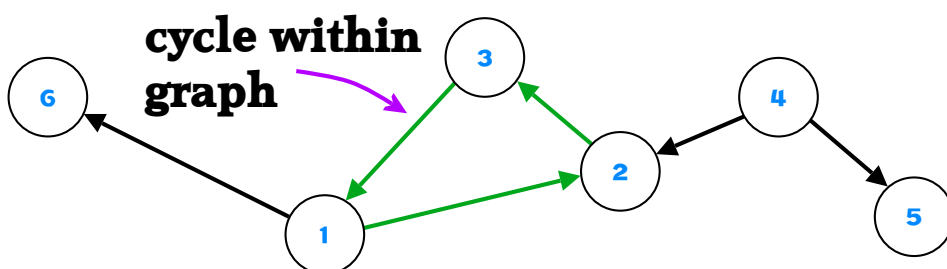


FIGURE 4.8: Diagram of a directed cyclic graph with a girth of 3. The vertex set $\{1, 2, 3\}$ is connected in a closed loop, beginning and ending at the same vertex.

The girth of a graph is defined as the length of the shortest cycle that exists within the graph [95]. For graphs that have no cycles (*acyclic graphs*), the girth is recorded as infinite.

4.3.12 Graph transitivity

Graph transitivity can be considered as a measure of clustering within a graph and is equivalent to the *clustering-coefficient* of a graph [96]. There are three measures of graph transitivity to consider: (1) global transitivity (2) local transitivity (3) average local transitivity.

NOTE: Graph transitivity should not be confused with the edge-transitive or vertex-transitive properties of a graph. These properties are related to graph automorphism rather than any notion of a clustering-coefficient.

Global transitivity is the ratio of closed triples in a graph to the total number of triples (see Equation 4.8). A *triple* is a subgraph of 3 connected vertices that exists within the graph (see Figure 4.9). A *closed* triple (also known as a *triangle*) implies that all 3 vertices in the triple are connected to one another. An *open* triple implies that 2 out of 3 vertices are connected. The total number of triples in the graph is simply the sum of open and closed triples.

$$\text{global transitivity } (C_G) = \frac{\text{number of closed triples}}{\text{total number of triples}} \quad (4.8)$$

Local transitivity measures the “cliquishness” of the local neighbourhood for a particular vertex v [97]. If v is connected to k_v neighbours, the total possible number of edges between those neighbours is $X_t(v) = \frac{k_v(k_v-1)}{2}$. Local transitivity is defined as the ratio of extant connections between neighbours $X_e(v)$ and total possible connections $X_t(v)$.

$$\text{local transitivity } (C_L) = \frac{X_e(v)}{X_t(v)} \quad (4.9)$$

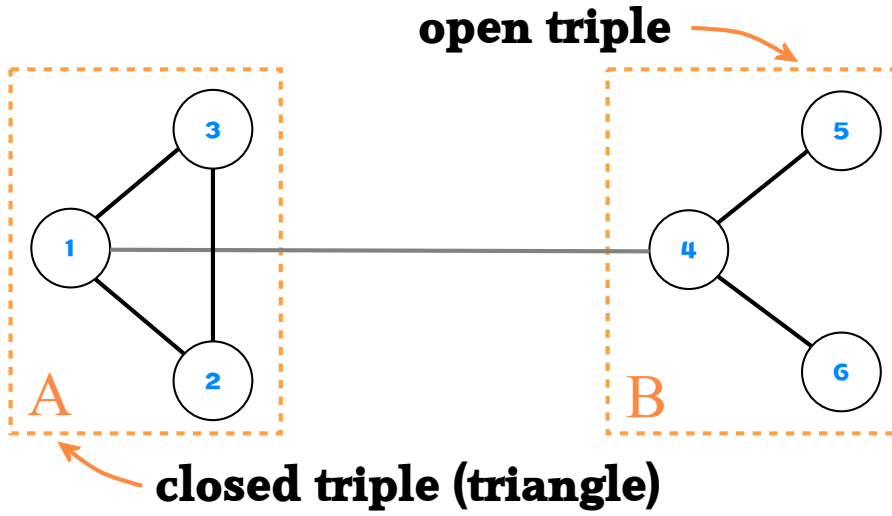


FIGURE 4.9: Diagram of a graph with a closed triple and an open triple highlighted. Vertex set $\{1, 2, 3\}$ in region A contains a closed triple (triangle) as there are edges $\{\{1, 2\}, \{2, 3\}, \{1, 3\}\} \subset E$. Vertex set $\{4, 5, 6\}$ in region B contains an open triple as there are edges $\{\{4, 5\}, \{4, 6\}\} \subset E$ whilst $\{5, 6\} \notin E$.

Average local transitivity \bar{C}_L is a global measure of transitivity in a graph. This is calculated by taking the mean of C_L for all vertices in the graph.

$$\text{average local transitivity } (\bar{C}_L) = \frac{1}{n} \sum_{i=1}^n C_L(i) \quad (4.10)$$

4.3.13 Connectivity

In a connected graph, *connectivity* considers the level of robustness in a network by measuring the minimum number of graph elements that need to be removed in order for the graph to become disconnected [98]. Thus there are two measures: (1) *vertex connectivity* evaluates the smallest subset of vertices that require removal for disconnecting the graph and (2) *edge connectivity* that considers edge removal. Graph connectivity can be computed by solving multiple *max-flow* problems that can be derived from the graph [99]. A max-flow problem is defined as calculating the maximum flow (based on edge weights as flow capacity) of notional information from a source vertex to a sink vertex. The igraph library has V^5 time complexity for vertex connectivity and V^4 for edge connectivity calculations.

4.4 Model property analysis

4.4.1 Methods

Model properties (described in Section 4.3) were collected for each model in the BioModels and curated models datasets (see Sections 4.2.2 & 3.2.1). 54 properties were generated for each of the reaction and species dependency graphs of a model. An additional property, *reaction graph stiffness ratio*, was also calculated bringing the total number of properties to 109. For some models certain properties were not possible to compute, e.g. when a division by zero occurred. I replaced all missing values with zeros. Nine model properties were found to be constant for all models and therefore would be of no use as performance indicators and thus were removed from the data set, resulting in 100 model properties available for analysis. By comparing the model properties of both datasets, I wished to assess whether the BioModels were representative of the curated models.

4.4.2 BioModel property correlation analysis

Roy proposes using *heatmap* visualisations as an important tool for network analysis in systems biology and other fields [81, 100]. Figure 4.10 presents a heatmap of the 100 analysed network property correlations for all 380 models in the BioModels dataset. Property correlation values were recorded as the *Pearson's correlation coefficient* (see Appendix A.1) which measures the linear relationship of each possible pair of property variables. The correlation heatmap has a thin diagonal line of single red points (positive linear correlation) where the same two properties are being compared and can be disregarded. There are also larger square block regions of red that are present on the diagonal. This is because multiple properties that have been generated using the same methods are grouped together on both axes. For example, *region A* shows that there are nine property values generated using the closeness centrality metric from the reaction dependency graph of the models. The

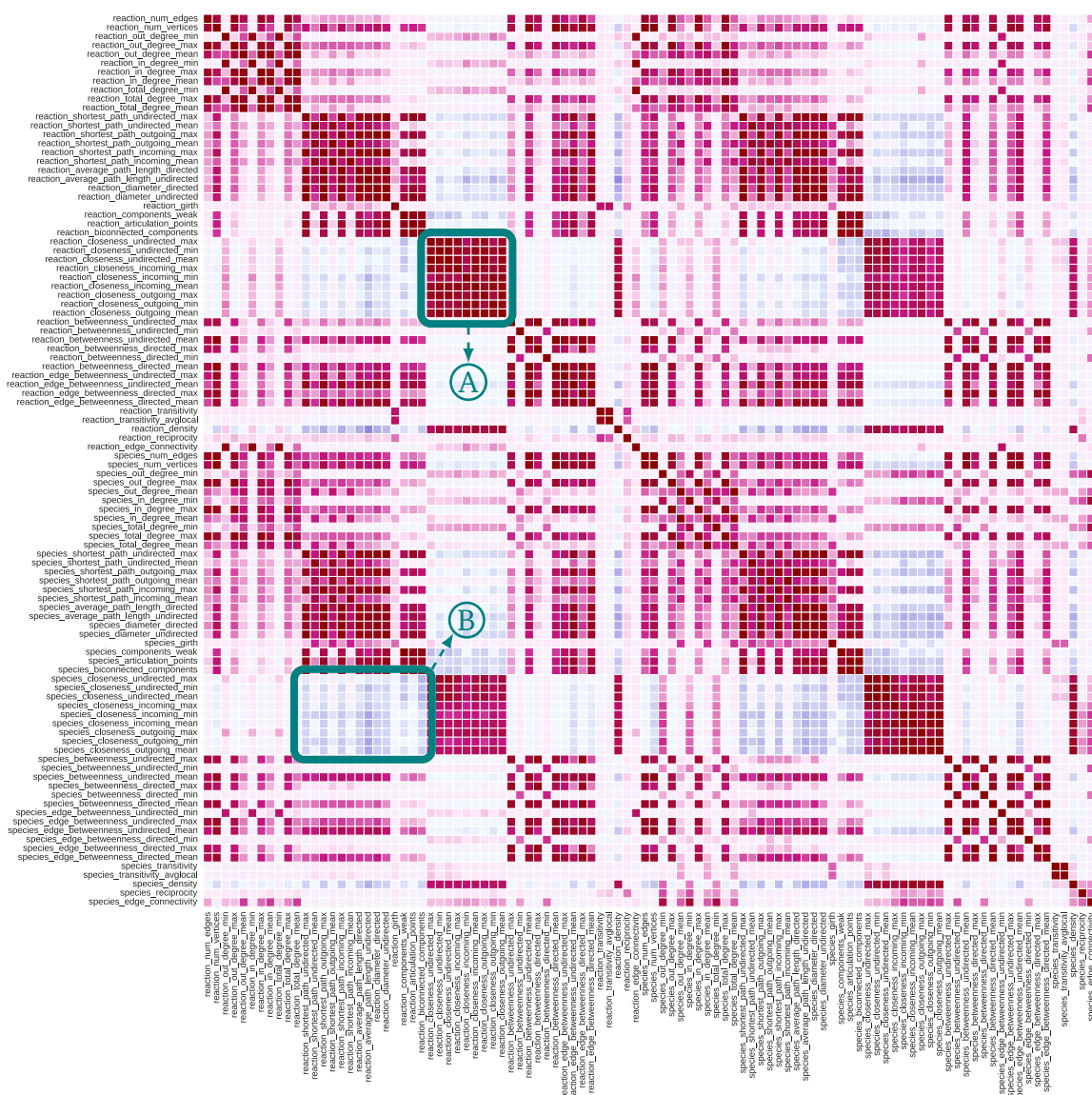


FIGURE 4.10: Heatmap of model graph property values correlations for the BioModels dataset. Both axes list 100 properties analysed from the reaction dependency graph and species network graphs in the same order. The heatmap values shown display the Pearson correlation values for each property-property combination over all 380 models in the BioModels dataset. Red values indicate positive correlations and blue values indicate a negative correlation, whilst whiter values indicate no/low levels of correlation. Vector version for high resolution viewing available at: <http://ssapredict.ico2s.org/ssapredict/static/analysis/propertyheatmap.pdf>

different variants of this metric vary the recorded edge directionality of the RDG and collects the max, min and mean values for each variant. One can observe that all of these property values are closely correlated. This raises the possibility that many of the properties within such a group may duplicate feature information (and thus some may be removed without losing feature information). Blue regions of the plot indicate a negative (i.e. inverse) linear relationship between property variables. The largest regions of inverse linear correlations exist between the closeness centrality

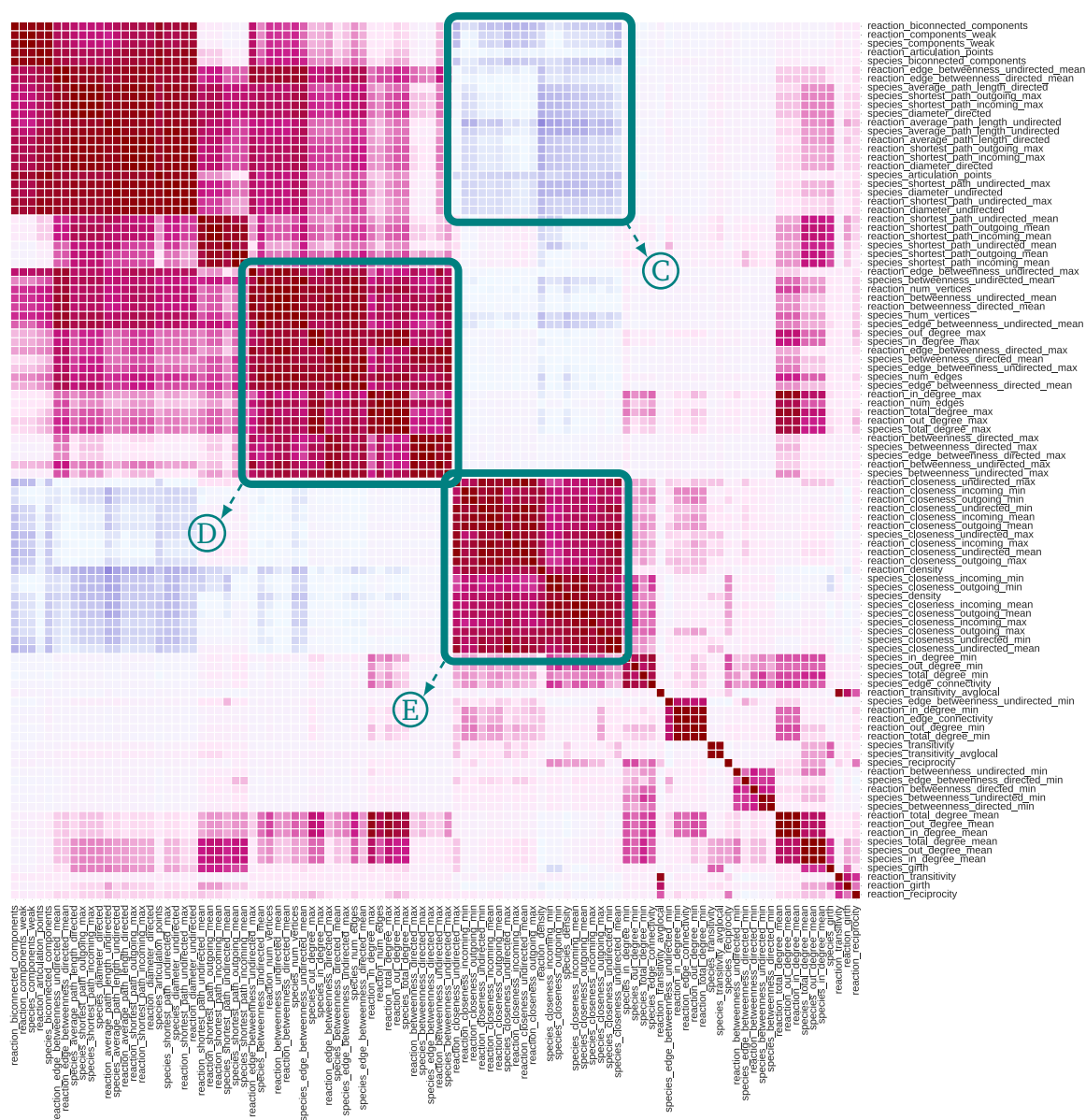


FIGURE 4.11: Hierarchically clustered heatmap of model graph property value correlations for the BioModels dataset. Both axes list 100 properties analysed from the reaction dependency graph and species network graphs in the same order. The heatmap values shown display the Pearson correlation values for each property-property combination over all 380 models in the BioModels dataset. Red values indicate positive correlations and blue values indicate a negative correlation, whilst whiter values indicate no/low levels of correlation. Vector version for high resolution viewing available at: <http://ssapredict.ico2s.org/ssapredict/static/analysis/propertycluster.pdf>

metrics and the shortest path length metrics (see *region B*). This corroborates the theory as closeness centrality is calculated using the inverse sum of shortest path lengths from a vertex v to all other vertices (see Section 4.3.9.1).

Figure 4.11 uses the same property correlation data as Figure 4.10, but uses hierarchical clustering on both axes. Clusters along the diagonal of this plot also reveal

areas of duplicated feature information (see regions *D* & *E*). The hierarchical clustering improves the visibility of feature redundancy between unrelated properties. Region *D* shows that the measures of betweenness is clustered with the number of edges, number of vertices and node degree metrics. One should note that betweenness measures are computationally expensive ($O(VE)$ time), whilst the other metrics are trivial to compute. Region *E* confirms that different variants of closeness measures are tightly clustered with one another, thus one only needs to compute a subset to extract full feature information. One can also see that density is clustered strongly with closeness measures. Density is trivial to compute ($O(1)$ time) compared to closeness measures, and this analysis reveals it may be possible to capture relevant feature information using just the density metric. Region *C* increases the scope of the findings of region *B* (from Figure 4.10) which showed that there is an inverse relationship between closeness measures and shortest path metrics. One can observe that the average path length, diameter, articulation points and components metrics are also inversely related to the closeness measures.

Clustering in this fashion not only demonstrates how strongly correlated pairs of arbitrary properties are, but also elucidates those that have a similar vector of correlation values for all n properties. Properties with very similar correlation vectors describe the same notional feature information, providing the opportunity to restrict the set of evaluated properties without compromising underlying feature data. This analysis reveals that within the domain of interest, there are strong feature relationships between different properties. For example, the number of vertices in the RDG is tightly clustered to the mean betweenness centrality of the RDG. This result is significant because it means that a property calculable in $O(1)$ time holds similar information to a property that requires $O(V(VE))$ time. In general, this analysis shows that a large number of computationally expensive measures are clustered with fast-to-compute measures.

NOTE: Whilst betweenness centrality is calculable in $O(VE)$ time, one must calculate it for every vertex to gain the *mean* value over all vertices. Therefore, the computation of this property has $O(V(VE))$ time complexity.

4.4.3 Dataset comparison and analysis using model topological properties

A quantitative analysis of model properties used the *Mann Whitney U* test (see Appendix A.2) to investigate whether both experimental model datasets considered in this thesis share the same distributions of values for a given property. This statistical test compared the distributions of values for each property of the BioModels and curated models datasets, given the null hypothesis that the distributions for both datasets are equal. The null hypothesis (distribution of values for a given property was equivalent) was rejected for 55 out of 100 properties ($p\text{-value} \leq 0.05$). This result is important because it demonstrates that the BioModels dataset does not comprehensively represent the fully specified curated models that I have sourced from computational biology literature. However, it does demonstrate that I have at least 45 properties that would be suitable for generating analysis that can be applied to fully parametrised “real world” models. Furthermore, the sample size for the curated models dataset (8 models) is small compared to the BioModels dataset (380 models) and thus the likelihood that the distributions would be equivalent is quite optimistic as the curated models values may themselves be outliers for a given property.

As I am interested in whether analytical insight from the BioModels dataset are applicable to the fully specified curated models, properties where curated models values lie within the range of BioModels values should still be relevant even if the distributions are not equivalent. A visual analysis was performed by comparing the distributions of values for each property in the BioModels dataset as black box-plots (including black points as outliers) against the curated models property values represented by coloured triangle points (see Figures 4.12 & 4.13). These properties are extracted from the topology of the *reaction dependency graph* (RDG and *species dependency graphs* (SDG) of the models. For properties that examine the minimum, mean or maximum value of a particular metric have been condensed such that multiple BioModels box-plots and curated models values are shown in a single plot. This analysis allows one to see where there are properties that the curated models

values lie outside of the range of the BioModels distribution and also properties for which the curated models would be considered outliers for the BioModels dataset.

Model size can be evaluated by observing the number of vertices for the RDG (number of reactions in the model) and SDG (number of species in the model). Figures 4.12 & 4.13 show that the number of vertices for curated models does lie within the range of BioModels values. However, one should note that both plots are on a logarithmic scale and the curated models values tend to be distributed toward the upper quartile of BioModels values. This indicates that the BioModels dataset overrepresents smaller models, a finding which is corroborated by the histogram of BioModels reaction network size (see Figure 4.1).

All of the plots that display the mean shortest path metrics for both RDG & SDG contain curated models values that lie outside of the range of BioModel values. Other properties which also have out-of-range curated models values are undirected average path length (RDG & SDG), mean directed edge betweenness (RDG), minimum directed edge betweenness (SDG), mean undirected edge betweenness (RDG) and minimum undirected betweenness (SDG). For these properties, the distribution of curated models values is higher than the distribution of BioModels values. This result can be explained by the presence of many smaller models in the BioModels dataset. Higher values for properties such as shortest path, average path length and betweenness also indicates that curated models may have reaction and species networks that are more strongly coupled than those for the BioModels dataset.

This analysis identifies the topological properties that have few good training examples for the BioModels dataset: edge connectivity (RDG & SDG) and girth (RDG). The low BioModel edge connectivity values indicate that the models are potentially “fragile” as removing a single edge can disconnect the dependency graphs. The fixed RDG girth value of 3 over all BioModels and curated models, indicates that every graph analysed has short cycles (indicating coupling), which is a finding that may bode well for PDM algorithmic performance.

There were also several other properties for which there was little variation in curated models values: articulation points (RDG), weakly connected components (RDG & SDG), girth (RDG) and biconnected components (RDG). Most curated model values (and the median BioModels values) for weakly connected components are 1 (with an upper quartile value of 5), indicating that these biosystems have a high level of interconnectivity and dependencies tend to be fully interconnected. The girth values for the SDG indicate coupling between species dependencies in these systems. The number of articulation points for the RDG is usually 1, indicating robust systems (i.e. with only one disconnection point). This insight is important when combined with the edge connectivity findings. Thus, it is likely that these systems are robust, but usually possess one critical “hub” node. However, one must not forget that as the majority of these models are small, this will negatively affect the prospect of finding multiple articulation points.

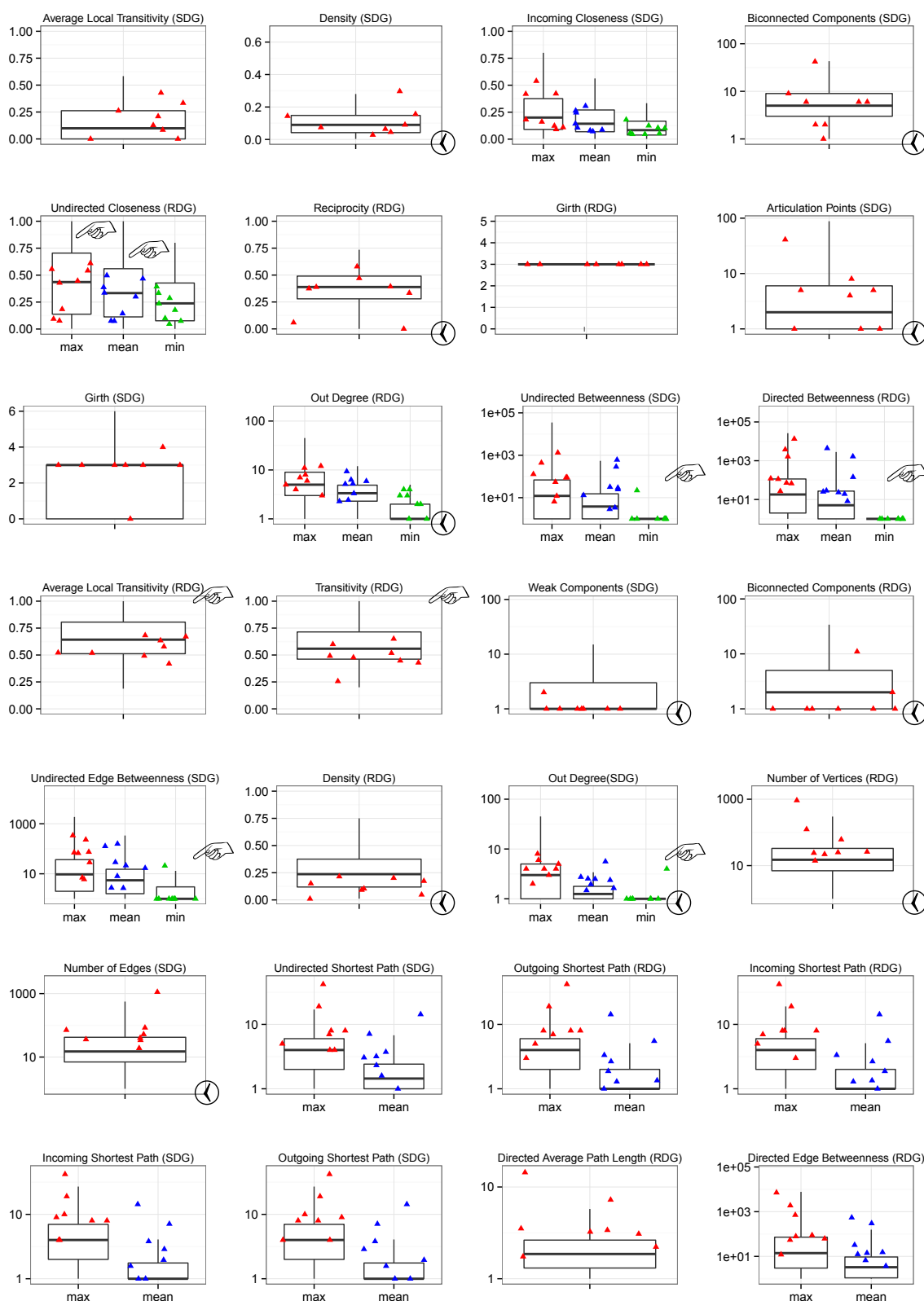


FIGURE 4.12: Property values and statistics of models in the BioModels and curated models datasets. The black box-plots show the property statistics for the BioModels dataset (this includes the black points which represent outliers). The horizontally jittered coloured triangles display the property values of models in the curated models dataset. The order of plots (row major) has been sorted by the p-value of the associated Mann-Whitney U test that compared the distributions of the curated models and BioModels values for each property, and where there were multiple subplots a mean p-value was used. Property types with computational complexity $\leq O(V + E)$ are denoted with a clock symbol at the bottom right corner of a plot. Specific properties selected by feature selection methods (see Section 6.4) are indicated by a pointing finger symbol. Vector version for high resolution viewing available at: <http://ssapredict.ico2s.org/ssapredict/static/analysis/property-distribution-plots.pdf>

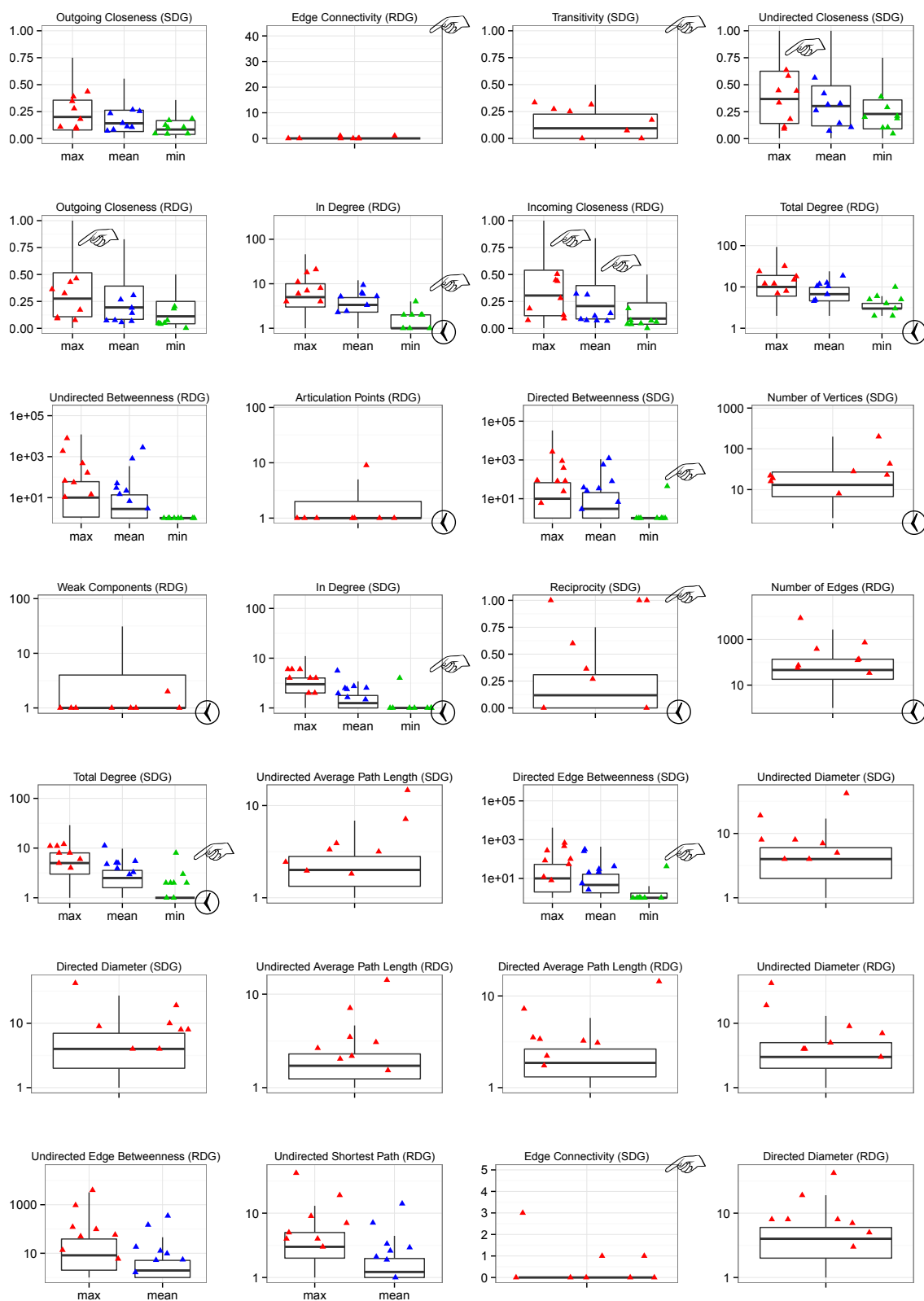


FIGURE 4.13: Property values and statistics of models in the BioModels and curated models datasets. The black box-plots show the property statistics for the BioModels dataset (this includes the black points which represent outliers). The horizontally jittered coloured triangles display the property values of models in the curated models dataset. The order of plots (row major) has been sorted by the p-value of the associated Mann-Whitney U test that compared the distributions of the curated models and BioModels values for each property, and where there were multiple subplots a mean p-value was used. Property types with computational complexity $\leq O(V + E)$ are denoted with a clock symbol at the bottom right corner of a plot. Specific properties selected by feature selection methods (see Section 6.4) are indicated by a pointing finger symbol. Vector version for high resolution viewing available at: <http://ssapredict.ico2s.org/ssapredict/static/analysis/property-distribution-plots.pdf>

4.5 Property computability and complexity

Figures 4.12 & 4.13 highlight properties that are “fast” to compute, as indicated by clocks in the bottom right corner of the sub-plots. These are properties that have time complexity that scales at most linearly with the size of the dependency graphs (and thus the size of the models). From Table 4.2 which lists the properties analysed and respective computational complexities, one can see that time scaling can be as favourable as $O(V^5)$. In practice, I found that once the size of a model reaches the order of 10^5 reactions, the more expensive properties require a large amount of time to be computed. In fact, if the simulation time of the large model executed is relatively sufficiently short, property calculations could greatly exceed the CPU time required by the simulation.

The third hypothesis of this thesis: *“An optimal selection of SSA can be made for an arbitrary model”* is closely related to the second hypothesis: *“There is a relationship between biomodel characteristics and SSA performance”*. To be explicit, one would use biomodel characteristics to determine the optimal SSA for an arbitrary model. If these hypotheses hold, any practical realisation of this relationship would depend upon a relatively fast evaluation of model properties. In other words, any tool based on these findings would need to determine the optimal algorithm in a timely manner to be of any significant value. An estimate of relative algorithm performance based on short “pre-runs” of available SSA implementations would be preferable to a slow property analysis. However, as demonstrated in Section 3.2.8, stochastic simulations are prone to transient algorithm performance variability. Therefore, a small “pre-run” does not guarantee an accurate selection of the fastest SSA for the entire duration of the simulation.

There are two steps that are required for the model property analysis: (1) dependency graph generation and (2) graph property generation. SDG generation is computationally trivial as there is a direct mapping from a stochastic model to its respective SDG. RDG generation requires an algorithmic method to calculate dependent reactions for each reaction in the system. The standard approach of generating the RDG

requires $O(M^2)$ time (see Algorithm 15). For large reaction networks, this $O(M^2)$ time scaling is undesirable. To remedy this, I have developed a $O(M)$ time complexity method for generating the RDG (see Section 4.5.1). Likewise, for tractable large model graph property analysis one should only consider the $\leq O(V + E)$ graph properties. Property correlations in Section 4.4.2 indicate that useful feature information of biomodel properties is duplicated in fast-to-compute and slow-to-compute properties. Thus, I may be able to remove some slow-to-compute properties without detrimentally affecting the quality of any analysis relating biomodel characteristics to algorithm performance.

4.5.1 $O(M)$ Reaction dependency graph generation

In this section, I introduce a novel algorithm to generate a RDG in linear time (see Algorithm 16). This can replace the typical “naive” $O(M^2)$ RDG generation that is found in established simulation software implementations [101, 102] (see Algorithm 15). My method works by pre-calculating a *species index (SI)* lookup table that stores reactions that depend on each species. When computing a particular reaction’s dependencies, I lookup the reaction’s affected species in the SI table to get the list of respective reaction dependencies.

One should note that my method still requires worst case $O(M^2)$ space in order to store the complete reaction dependency graph. Indurkha and Beal [48] have also introduced a dependency graph that provides $O(M)$ generation time, but also offers $O(M)$ space complexity. Their *bipartite dependency graph* uses the same approach as my $O(M)$ RDG generation method, but instead stores the SI lookup and affected species intermediates. When a particular reaction dependency set is required, the full list for a single reaction can be generated from these intermediates. This is effectively a trade off between algorithm performance and space - however this algorithmic cost is quickly amortised for larger models. Therefore, the Indurkha-Beal method can be considered the state-of-the-art with regard to SSA dependency graphs.

Algorithm 16 Fast RDG Generation ($O(M)$)

```

1: procedure FASTRDG(reactions)
2:   ▷ initialise
3:   store list of reactions  $RL$   $[1..M]$ 
4:   create empty dependency graph  $DG$ 
5:   create empty species index table  $SI$   $[1..N]$ 
6:   ▷ generate species index table
7:   for  $i \leftarrow RL[1..M]$  do
8:     for  $j \leftarrow reactants_i$  do
9:       ▷ append  $i$  as affected by species  $j$ 
10:       $SI_j \leftarrow i$ 
11:     end for
12:   end for
13:   ▷ now generate dependency graph
14:   for  $i \leftarrow RL[1..M]$  do
15:     for  $j \leftarrow affected_i$  do
16:       ▷ reactions indexed by  $j$  are dependencies of  $i$ 
17:        $DG_i \leftarrow SI_j$ 
18:     end for
19:   end for
20: end procedure

```

4.6 Summary & conclusions

This chapter introduced a methodology for characterising biochemical models of the type that can be executed by stochastic simulation algorithms. I have adopted a technique used for systems level biology: *network analysis*. Some SSA variants (e.g. NRM, ODM) use dependency graphs to boost computational performance. Mathematical graphs are analogous to networks, and can be quantitatively assessed using graph topological properties. Due to a lack of fully parametrised biochemical models, network analysis is performed upon unweighted graphs. The analysis does not take reaction rates into account (i.e. an extremely rare event carries as much weight as a commonly executed reaction). Furthermore, a topological (structural) analysis of a model does not consider transient variability due to differing states reached during simulation.

Chapter 3 introduced the concept that algorithm performance is related to model characteristics. It was shown that algorithm performance profiles fall into groups or “classes” of model. Using the data generated from the graph topological analysis of

the models, I can begin to address the second and third hypotheses of this thesis. To comprehensively evaluate these hypotheses, I first need to collect comprehensive benchmark data for the BioModels dataset. SSA benchmarking is addressed in the following chapter.

Chapter 5

Benchmarking Stochastic Simulation Algorithms

5.1 Introduction

Many published SSAs are tested with an insufficient number of models, mostly tailored to properties of the newly introduced algorithm. Consequently, it is hard to extrapolate or compare performance between algorithms as each will often be benchmarked against competitors' algorithms using only favourable models. To address this issue, I have created a performance benchmarking suite which allows for a direct and unbiased comparison of stochastic simulation algorithms. The benchmark suite provides reference implementations of 9 different SSAs and a set of 380 test models. Aside from analysing computational performance, I have checked the statistical correctness of the algorithms using the *Discrete Stochastic Models Test Suite* [103].

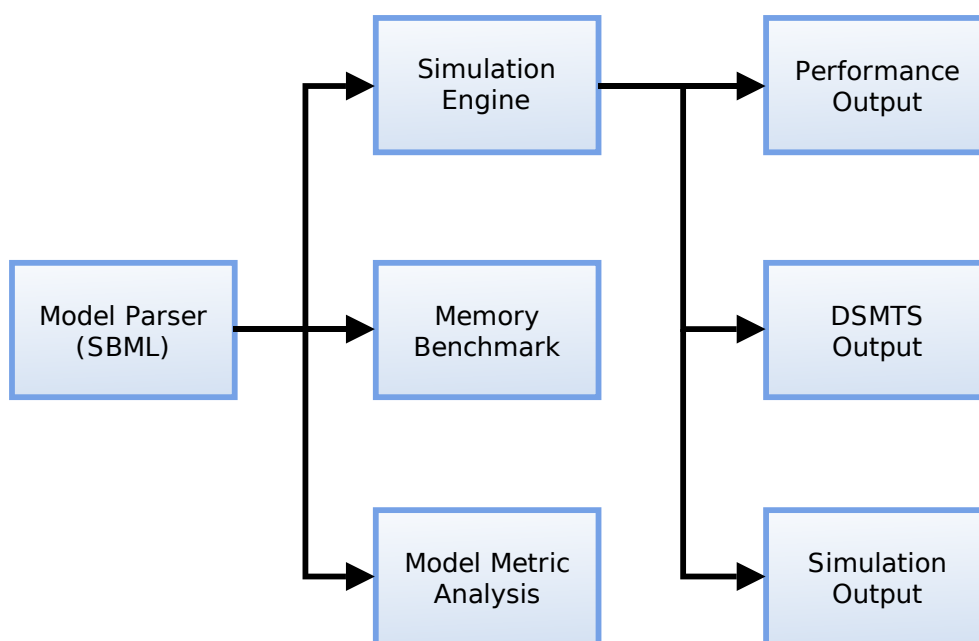


FIGURE 5.1: Overview of benchmarking suite

5.2 Benchmarking suite

5.2.1 Overview

The benchmarking suite is designed to be a multi-purpose tool for researchers using stochastic simulation algorithms to simulate biological reaction networks. A schematic representation of the structure is shown in Figure 5.1. Users supply their stochastic models in SBML format [33]. Software developers are able to implement their own algorithms and test them in the suite against other implemented algorithms. Furthermore, developers can use the source code for the supplied algorithms in their own software. The benchmarking suite is released under the terms of GNU General Public License (version 3 or later).

5.2.2 Algorithm implementations

Algorithms were implemented in the benchmarking suite with as little deviation as possible from the descriptions in the literature. The code was written in C++ in an object oriented style, with care taken to ensure good performance. Common interfaces and encapsulation were used to allow new algorithms, model loaders, output methods, random number generators and timers to be easily integrated into the suite. Models are initially parsed using the Infobiotics Workbench multi-compartmental stochastic simulator SBML model loader [21, 52], and “flattened” so they can be simulated by standard stochastic algorithms. The details of the experiment to be performed is dictated by a XML parameters file. Different parts of the experiment are timed independently so that the user can analyse different aspects of algorithmic performance, for example how initialisation times vary by algorithm.

Algorithm	Type	Ref
Direct Method	Exact	[31]
First Reaction Method	Exact	[10],[31]
Next Reaction Method	Exact	[25]
Optimised Direct Method	Exact	[26]
Sorting Direct Method	Exact	[27]
Logarithmic Direct Method	Exact	[28]
Partial Propensity Direct Method	Exact	[29]
Composition Rejection	Exact	[30]
Tau Leaping	Approximation	[11]

TABLE 5.1: Summary of available stochastic simulation algorithms in benchmarking suite.

Algorithms were tested against a subset of the Discrete Stochastic Model Test Suite (DSMTS) [103] test models in order to validate that they had been implemented correctly and to ensure the benchmarking suite was generating accurate results (see Section 5.3). The test suite checks the statistical correctness of the output by comparing the mean and the standard deviation of a simulator’s outputs to its own collection of verified results for the same model. Algorithms were also tested with models referred to in the results section of their own papers in order to verify that

the performance of my implementations tally with the authors' original results. This allowed me to check that I had not inadvertently produced an under-performing implementation of each algorithm.

After implementation and accuracy testing, I found that certain algorithms had reproducibility issues for their respective papers. For example, the original PDM paper had a typographical error in the algorithm listing which resulted in incorrect simulation trajectories [29]. This fault was reported by the authors who subsequently made errata available [104]. Another issue for SSA implementation reproducibility is a lack of algorithmic detail in some literature. The CR algorithm paper only provides a qualitative description of the algorithm, along with expected computational complexity and benchmark results for large random reaction networks [30]. Producing an implementation that replicated the CR paper benchmark results took a large amount of development time.

5.2.3 Benchmark models

The benchmarking dataset contains 380 models in SBML [33] format retrieved from the *BioModels database* [85]. As described in Section 4.2.2, the BioModels have stochastic reaction rates set to a fixed value of 1.0 due to the scarcity of curated stochastic models. To complement the topological analysis detailed in Chapter 5, the species amounts for all the BioModels are set to a constant 100. This removes the transient variability that can affect algorithm performance (see Chapter 3), as the static property analysis does not capture this element of a simulation.

The other dataset of this thesis is the curated models dataset (see Section 3.2.1). The smaller set of curated models were described and performance benchmarked in Chapter 3.

5.3 Simulation algorithm accuracy testing

Stochastic simulation algorithms are by their very nature non-deterministic, which presents difficulties for detecting design or implementation errors. Standard development practices can be applied such as regression testing and unit testing by using fixed PRNG seeds. This ensures simulation algorithm behaviour is both deterministic and reproducible. However, what happens if the PRNG implementation itself changes? Precise PRNG output may vary from version to version and can conceivably differ by platform. Furthermore, different SSA variants often consume random numbers in radically varying amounts and requirements. For example, NRM consumes one random per iteration, whilst CR consumes an unbounded number of randoms due to rejection sampling.

A different approach is required to evaluate SSAs: probabilistic comparison. Evans et al. introduced DSMTS in 2007 [103] for statistically checking the accuracy of SSA implementations. The stochastic test suite has been adopted by established simulation software such as COPASI [101] and Systems Biology Workbench [83]. DSMTS provides 36 model files in SBML format, along with “gold standard” time-series and respective standard deviation values for each model. These gold standard values have been computed either analytically or numerically for each model variant. These 36 model files are minor variants of 3 simple biochemical systems. Many of the tests are of identical biosystems but only vary in their usage of SBML features. Thus, DSMTS also places a strong emphasis on testing the quality of the implementation of a model parser and simulator features.

Figure 5.2 shows the gold standard time-series and concomitant standard deviation traces for model dsmts-001-01. This model is a variant of a *birth-death process* [105]. The system only possesses a single species X with initial amount 100. There are two reactions in the system: (1) *birth* $X \rightarrow X + X$ with stochastic rate 0.1 and (2) *death* $X \rightarrow \emptyset$ with stochastic rate 0.11. The system is simulated for 50 seconds of

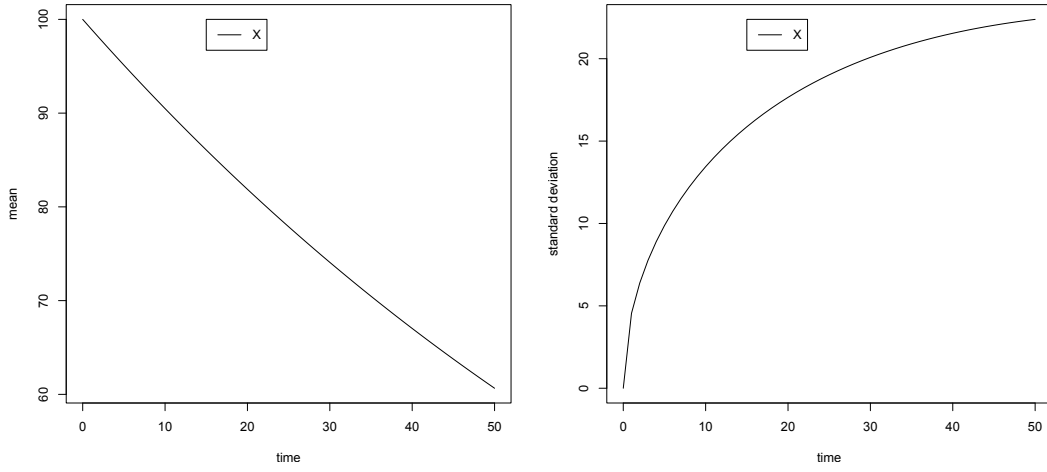


FIGURE 5.2: DSMTS birth-death process model (dsmts-001-01) “gold standard” mean and standard deviation results. These results were generated analytically, and are used to test simulator results by comparing distributions.

simulation time, and because the death rate exceeds the birth rate the amount of X decreases.

In order to test a simulator implementation, a developer must run the model for the specified 50 seconds simulation time and log species amounts at every one second interval. Evans et al. recommend performing a minimum of 10,000 simulation runs to attain a sample size that is adequate for statistical testing. A Z -test must be performed to assess the null hypothesis that the simulator is generating a valid trajectory. X_t is the random variable representing the species X at time t , thus $\mu_t = E(X_t)$ and $\sigma_t = \sqrt{Var(X_t)}$. After application of the Central Limit Theorem it follows that [103]:

$$Z_t \equiv \sqrt{n} \left(\frac{\bar{X}_t - \mu_t}{\sigma_t} \right) \sim N(0, 1) \quad (5.1)$$

where $\bar{X}_t = \frac{1}{n} \sum_{i=1}^n X_t^{(i)}$ when $X_t^{(i)}$ is the value of X_t for the i^{th} simulator run. Evans et al. report that Z_t values should lie within the range $(-3, 3)$. Thus values that lie outside this range should be considered a violation of the null hypothesis. The standard deviation test is computed as follows:

$$Y_t \equiv \sqrt{\frac{n}{2}} \left(\frac{\hat{S}_t^2}{\sigma_t^2} - 1 \right) \sim N(0, 1) \quad (5.2)$$

where $\hat{S}_t^2 \equiv \frac{1}{n} \sum_{i=1}^n (X_t^{(i)} - \mu_t^{(i)})^2$. The specified range for non-violation of this test is $(-5, 5)$. As the test runs are probabilistic, a perfectly valid simulator may produce trajectories that violate some of the 50 mean and standard deviation tests per dsmts model. Therefore, the authors state in supplementary materials that it would not be unreasonable to expect a valid simulator to fail up to 3 mean tests and 6 standard deviation tests when 10,000 runs are evaluated.

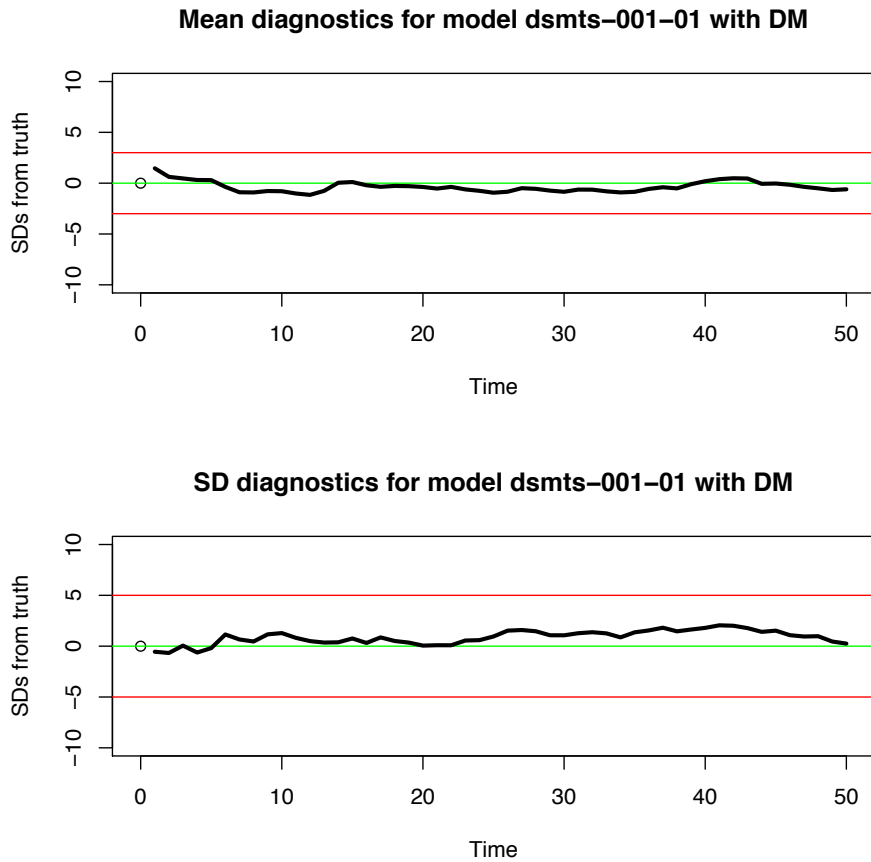


FIGURE 5.3: DSMTS mean and standard deviation test for DM using model dsmts-001-01. The null hypothesis of this test is that the SSA generated a valid set of trajectories. Red line limits indicate the error range which when crossed indicate a violation of the null hypothesis. 10,000 runs were computed and no errors detected.

Figure 5.3 shows the accuracy of the DM algorithm for the birth-death process model dsmts-001-01 described earlier. One can observe that the Z values for the mean

diagnostics all lie within the range $(-3,3)$ which means there are no violations of the null hypothesis. Likewise, the standard deviation test values over all values of X_t lie within the range $(-5,5)$, so I can conclude that my DM algorithm implementation is performing valid simulations for this test. Figures 5.4 & 5.5 show the accuracy results with *dsmts-001-01* for the other 8 algorithms that I have implemented and benchmarked. Since there are no violations of the null hypothesis for any of these algorithms with this test, one can be confident in the simulation accuracy of the SSA implementations.

5.4 Measuring algorithm performance

The performance metric used to measure algorithmic computational speed was reactions per second (*rps*) of CPU time. *Rps* allows one to compare algorithm performance in a manner that ignores simulation run time. This means that algorithm performance can be compared between two models that take vastly differing amounts of time to execute. Using *rps* also improves comparative accuracy: if one wishes to run an algorithm for x seconds, and measure how many reactions are executed, the amount of time elapsed would almost certainly not be *exactly* x seconds, but a number very close to x seconds. If this was compared to another run of x seconds, neither run would be exactly the same amount of time, and thus a comparison in this manner would lose accuracy. Dividing the number of reactions executed by the exact simulation time to get a result in *rps* generates a value that is appropriate for comparison.

All runs were executed on a single core of an otherwise idle benchmarking machine that possessed an Intel i7 2600K CPU with 16GB RAM and was running Ubuntu 11.04. The large amount of RAM available and size of models involved meant that all simulations could be run in memory and thus avoid performance deterioration caused by memory paging. As the BioModels have fixed parameters and species amounts, there are no transient shifts in model-algorithm performance. Therefore, an accurate measure of model-algorithm performance can be achieved with a short

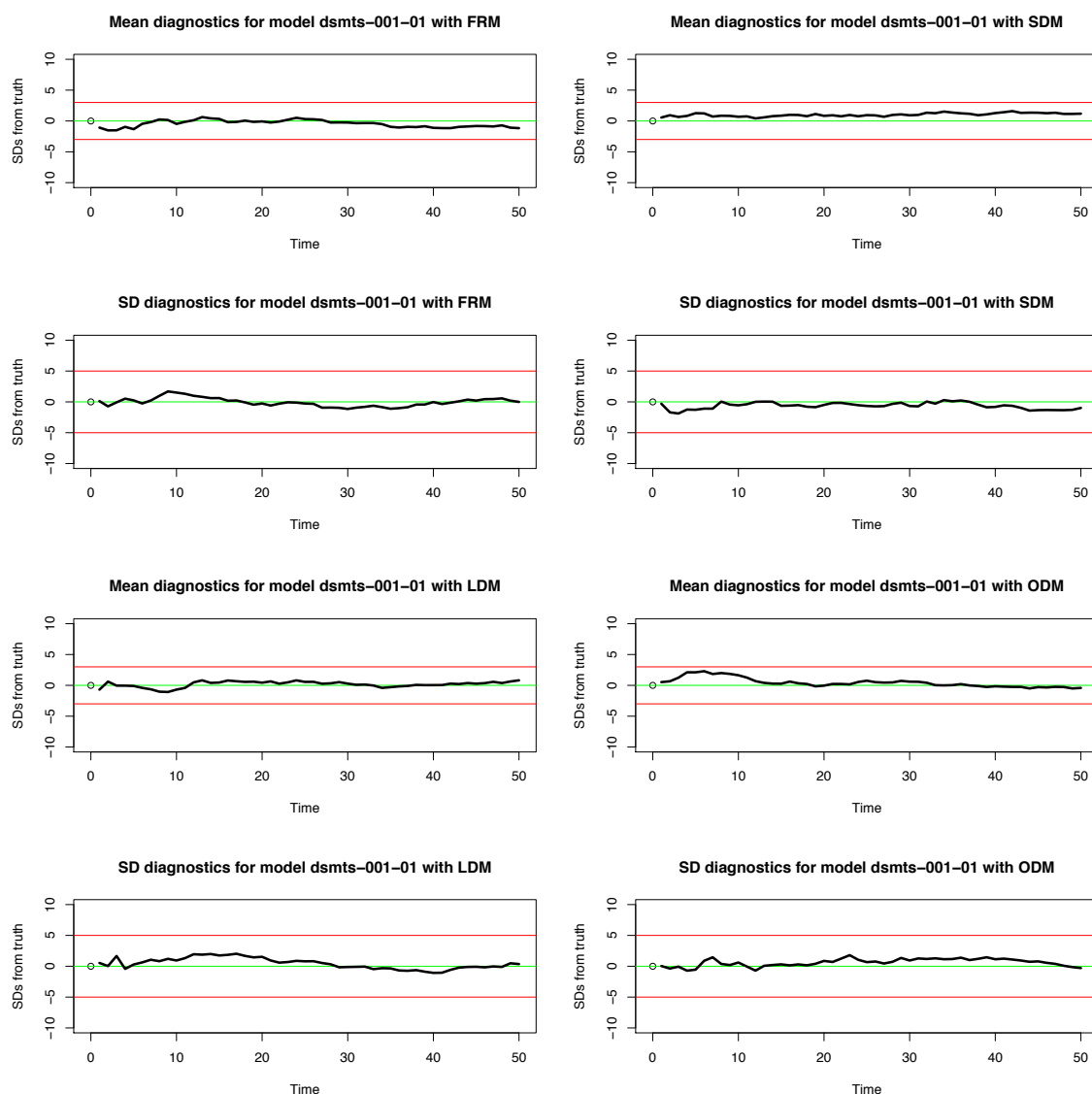


FIGURE 5.4: DSMTS mean and standard deviation tests for FRM, LDM, SDM & ODM using model dsmts-001-01. The null hypothesis of this test is that the SSA generated a valid set of trajectories. Red line limits indicate the error range which when crossed indicate a violation of the null hypothesis. 10,000 runs were computed and no errors detected.

simulation benchmark run. Each of the BioModels was executed for 10 seconds of CPU time for all 9 algorithms. Each algorithm was run 10 times on each model, hence a total of 90 *rps* values for each model. 10 seconds of CPU time for each model/algorithm combination would be enough to determine an accurate result for algorithm performance.

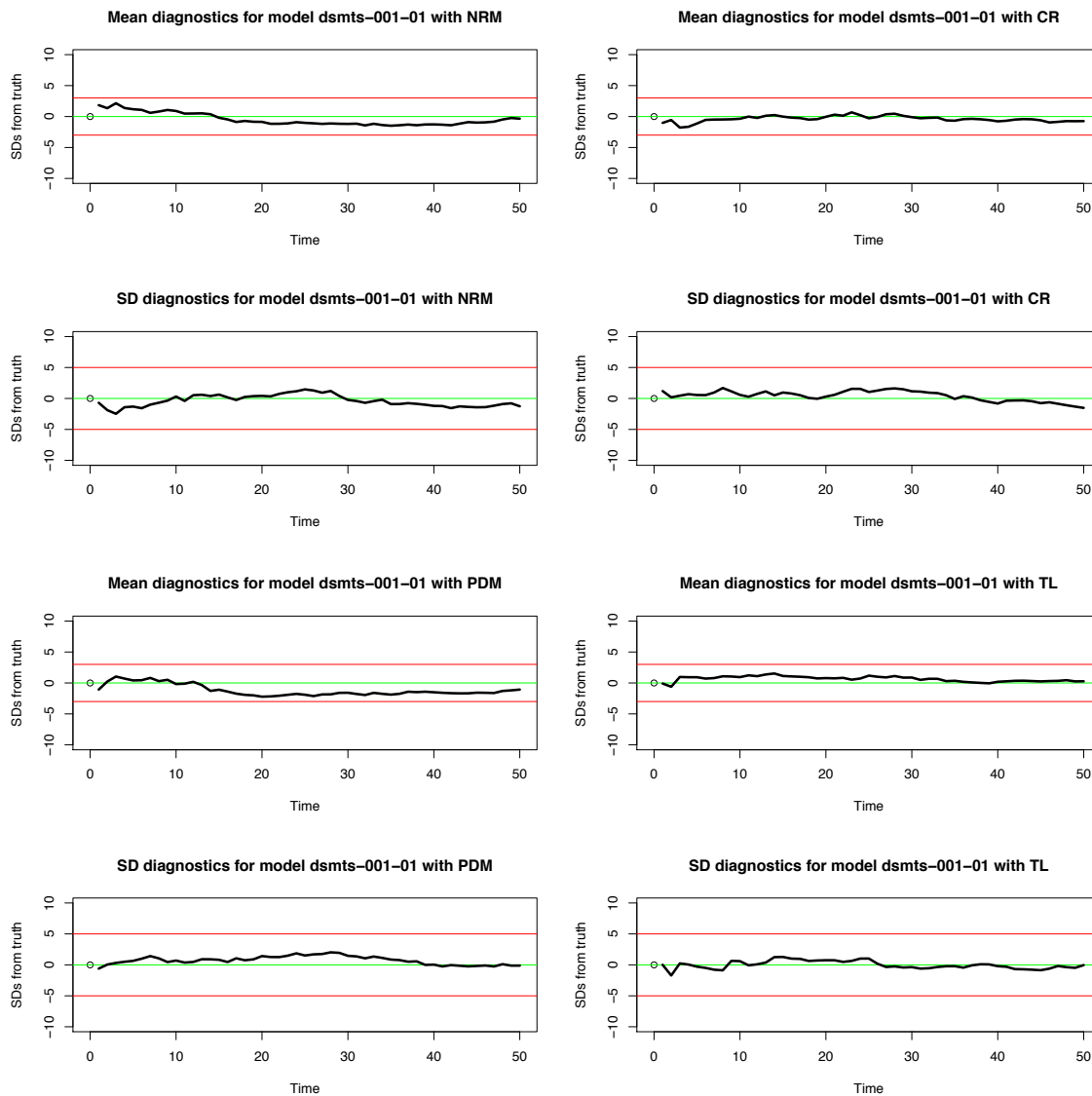


FIGURE 5.5: DSMTS mean and standard deviation tests for NRM, PDM, CR & TL using model dsmts-001-01. The null hypothesis of this test is that the SSA generated a valid set of trajectories. Red line limits indicate the error range which when crossed indicate a violation of the null hypothesis. 10,000 runs were computed and no errors detected.

5.5 Preliminary BioModels performance analysis

In Figure 5.6, the histogram displays the number of times a particular algorithm was considered the fastest algorithm (highest mean *rps*) for each model in the BioModels dataset. This metric (fastest algorithm) is an overview of algorithm performance that represents my goal of selecting the fastest performing algorithm. However, it does

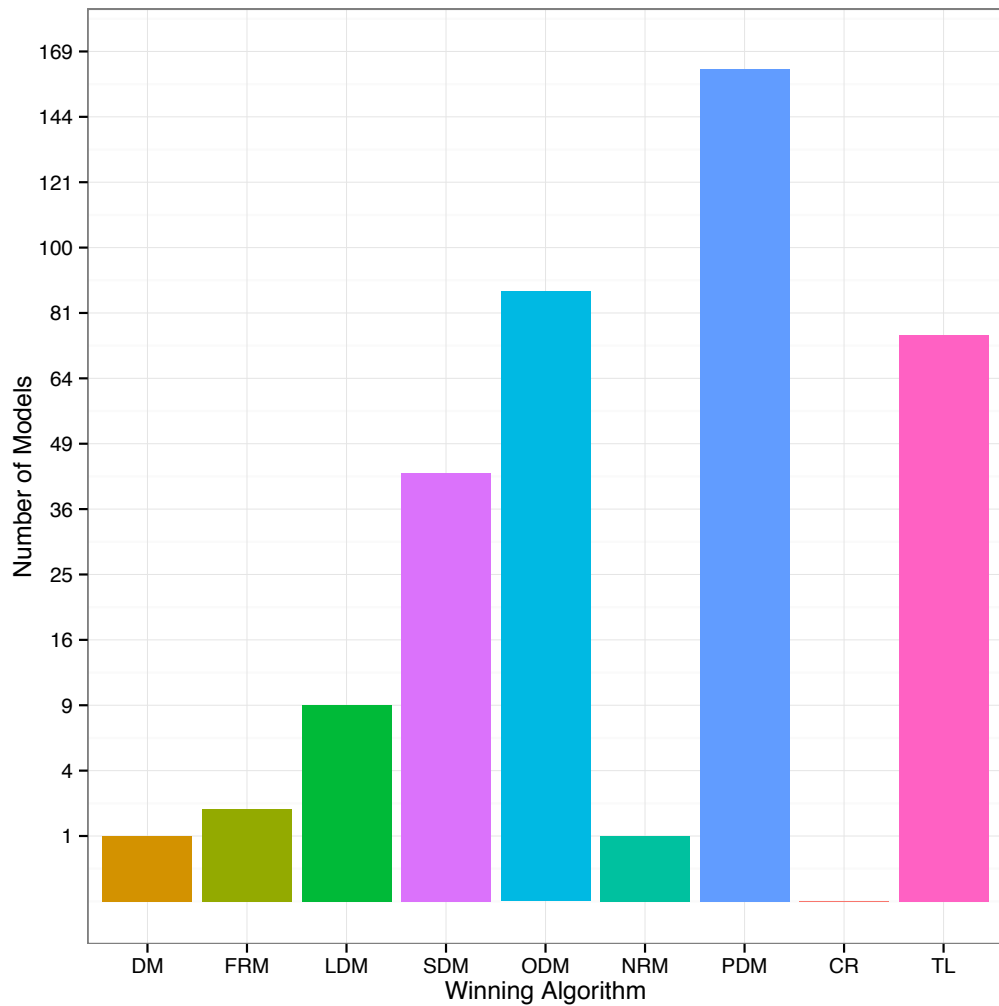


FIGURE 5.6: Histogram displaying the number of times a particular algorithm was classed as being the fastest algorithm (highest mean *rps*) when executed on every model in the BioModels dataset. The y-axis is on a square root scale.

not account for an algorithm consistently performing well yet not necessarily being the absolute fastest algorithm on many models. To put this result into perspective, I compared the performance of each algorithm to the best algorithm in the group for each of the models. Figure 5.7 shows that three frequent winners PDM, ODM and SDM, have very similar performance profiles (they are all improved variants of DM) with the notable exception of a few models for which PDM performs badly. TL, another algorithm in the top 4, performs exceptionally well for about 20% of the models, but is outperformed by other algorithms for the rest of the dataset. For the worst performers, CR and FRM, there is a clearly visible gap that separates their performance from the best.

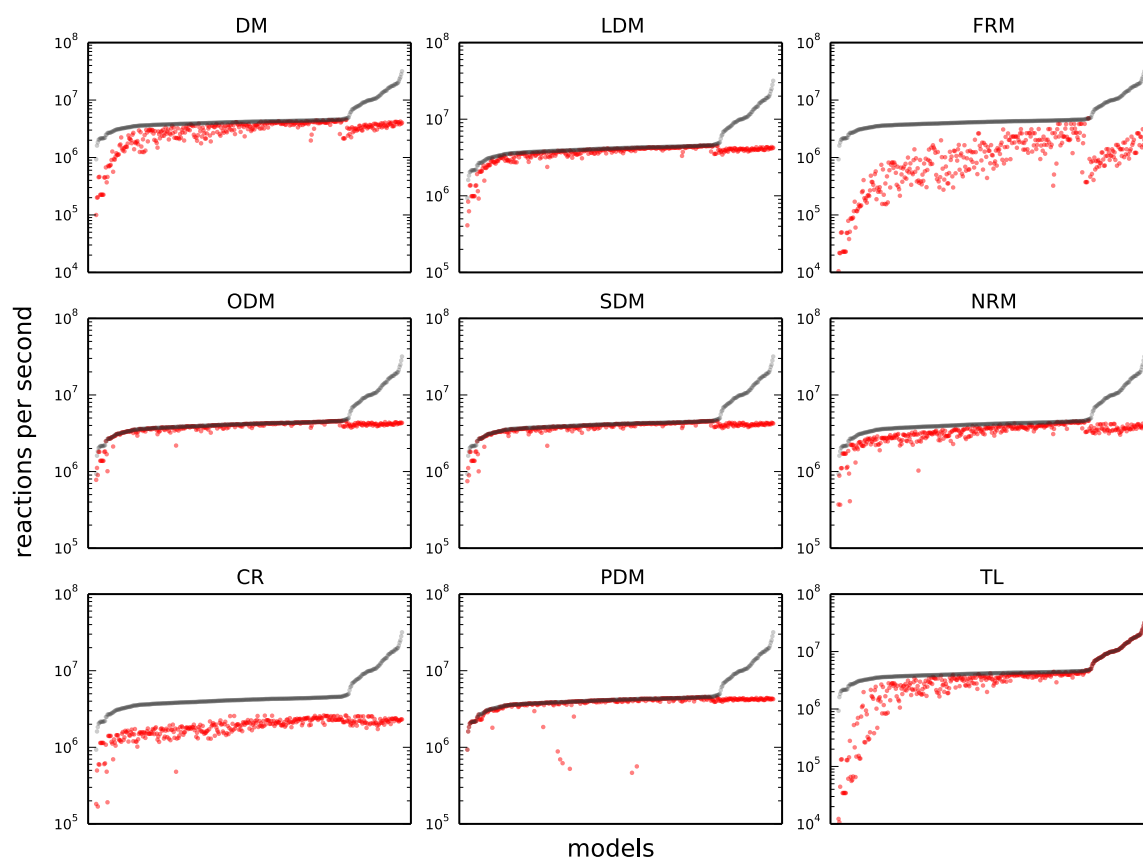


FIGURE 5.7: Comparison of the performance of each algorithm against the best algorithm performance for every model in the BioModels dataset. The red data points are the algorithm performance values for each model. The grey data points show the best performance for each model. The models on the horizontal axis are ordered by best performance.

Table 5.2 shows how consistent the top 4 winning algorithms are. For each of those algorithms, I measured how many times it was ranked below the top 4. ODM was the algorithm that most consistently remained in the top 4 (378 out of 380 models), but was closely followed by SDM (368 out of 380 models). On the other hand, PDM and TL were ranked below the top 4 many times, including being ranked as the worst algorithm for some models. TL in particular remained in the top 4 for only 80 models.

Figure 5.8 shows a bi-clustered heatmap of algorithm performance for each model in the BioModels dataset, with each heatmap cell containing a normalised algorithm *rps* result for a given model. This complements the histogram in Figure 5.6, allowing one to observe the performance of an algorithm over all models. One can deduce

rank	ODM	SDM	PDM	TL
> 4	2	12	42	300
> 5	0	1	22	287
> 6	0	0	17	205
> 7	0	0	7	51
= 9	0	0	6	8

TABLE 5.2: Number of times one of the 4 best performing algorithms (See Table 5.3) was ranked below the top 4 for any of the 380 models from the BioModels dataset. Each row shows the total number of models for which an algorithm was ranked under a given threshold. The lowest possible rank was 9.

how similar the performance profiles of algorithms are to one another using the clustering analysis.

Both these analyses highlight some interesting results that may initially seem unexpected. Strikingly, one of the most advanced algorithms, CR, fares poorly in that it cannot be classed as the fastest algorithm for any model (see Figure 5.6). In Figure 5.8, one can observe that CR performs stably across all models, but as a relatively advanced algorithm, one might be surprised that its performance is generally low relative to other algorithms. Indeed, CR is much slower than DM on most models (even though it is based upon it), in spite of the benefits of a reaction dependency graph and composition-rejection sampling. The reason for this poor recorded performance is likely to be caused by the model sizes available in the dataset. As shown in Figure 4.1, only a small number of “large” models exist in the dataset, and the overheads introduced by composition and rejection sampling outweigh the performance benefits obtained when used with small models. One can assume that even the largest models in the BioModels dataset do not cross the threshold of size that allows CR to really show a performance advantage over the other algorithms. This first result does highlight the fact that the state of the art SSA is not necessarily performant for any given model.

Another interesting result is that FRM, which one might have assumed to be the worst performing algorithm across all models, is the fastest algorithm for more BioModels than CR, DM or NRM. Looking at Figure 5.8, one can see that FRM is indeed the

slowest performing algorithm for the vast majority of models. The model performance vector for FRM mostly contains dark or red (low) performance values. Even in its green (high performance value) regions, FRM is slower than many of the other algorithms, and some algorithms e.g. ODM and SDM outperform this algorithm for almost every model from visual inspection. The fact it performs fastest on certain models is an indication of the presence of tiny reaction networks (e.g. 1 or 2 reactions) in the BioModels dataset. With models of this size, performance overheads are incurred for more advanced algorithms without achieving any performance benefits. In this situation, FRM would have low random number usage as it requires one random number per reaction in the network, and has a combined step for reaction selection and τ calculation.

TL is another algorithm that has the best computational performance for a large number of BioModels. One can see from Region A of Figure 5.8 that TL has a number of low performance values, whilst Region D shows large clusters of strongly pronounced high performance values. Region D contains small & simple reaction networks that allow TL to apply many reactions per algorithmic iteration. Considering that the performance values have been normalised on the logarithmic scale, this means TL actually performs orders of magnitude faster than other algorithms for some models. However, one can also observe from Region A that TL is amongst the slowest algorithms for other models. Region A contains models with large reaction networks, indicating that TL does not scale well with larger models. This result highlights that in a principled selection of SSAs, TL would be an important algorithm that has excellent performance with some models, but needs to be replaced by a different algorithm for models it struggles with.

In Figure 5.8, one can see that the performance profiles for ODM, SDM and LDM are similar in pattern. This result is quite expected as all three algorithms are closely related; they are all based on DM and use a reaction dependency graph. The only difference between them is that they each have different methods of reducing the search depth in the reaction selection step of the algorithm (see Sections 2.4.5, 2.4.6 & 2.4.7). The difference between ODM and SDM in the heatmap

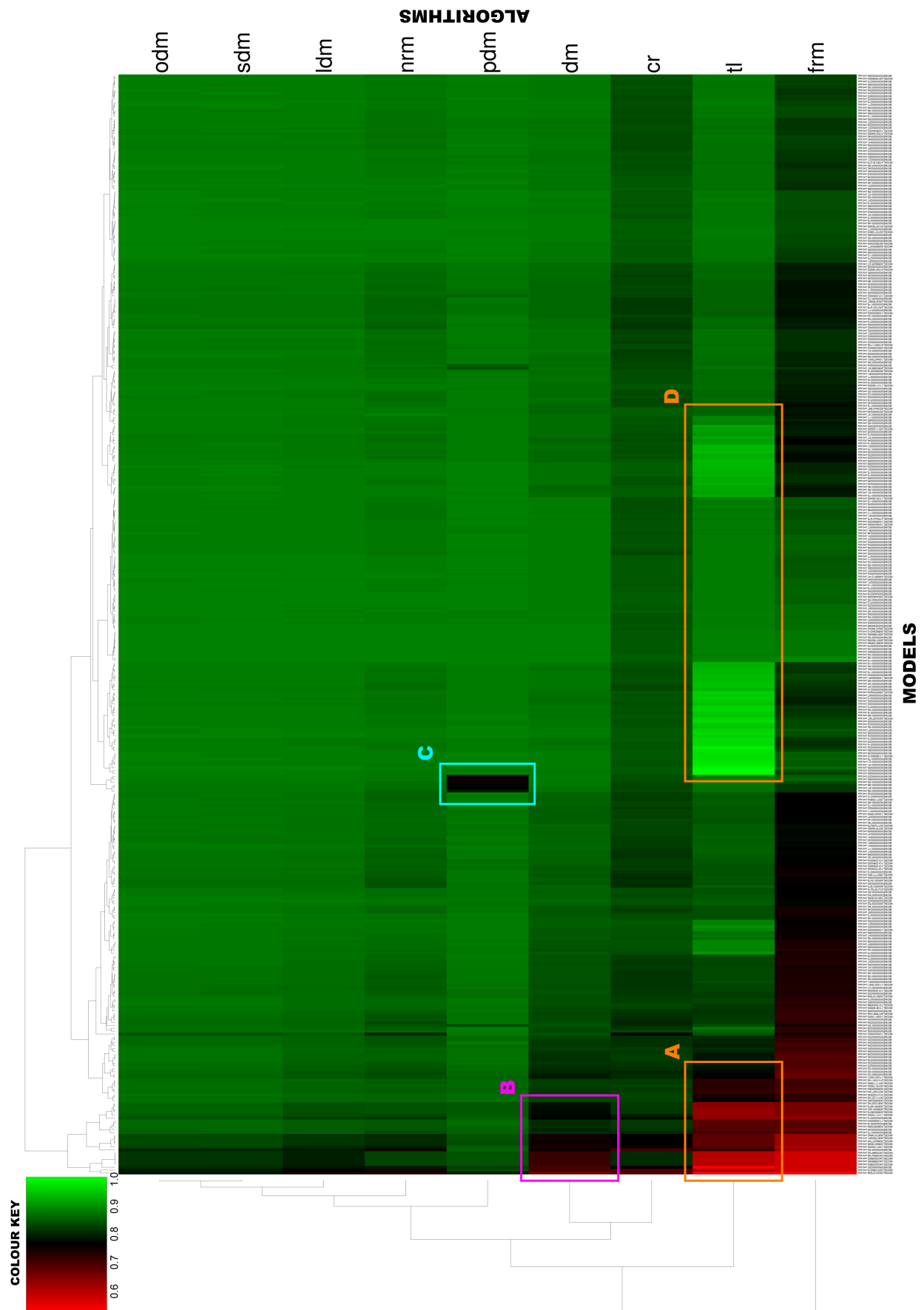


FIGURE 5.8: Bi-clustered heatmap showing the performance of all 9 algorithms for every model in the BioModels dataset. Logarithmic scaling was applied to the performance values and each vector of values was normalised to the range $[0, 1]$.

is almost imperceptible, but with closer inspection one can see that LDM is slightly slower than ODM or SDM. This is confirmed in Figure 5.6, as LDM is far less frequently the fastest algorithm when compared to SDM or ODM. One can see from this histogram that ODM outperforms SDM with this metric, even if the performance bi-cluster heatmap and dendrogram indicate little difference between them. It is important to note that SDM's advantage over ODM is that it can optimise for transient shifts in the propensities of a model. The BioModels analysis has constant propensity values which favours ODM over SDM. Even with this disadvantage, Figure 5.8 indicates that the performance differences are almost negligible. Thus, I can hypothesise that with complete models SDM would actually be the most performant of these two algorithms.

NRM is clustered quite closely to ODM, SDM and LDM in Figure 5.8, but overall has a slower (darker) performance vector. This decreased performance is confirmed in Figure 5.6, as NRM is only the fastest algorithm for a single BioModel. NRM is different to ODM, LDM and SDM in that it is based on FRM rather than DM, and also uses more complex data structures compared to the other algorithms. As discussed by Cao et al. [26], the overhead of these data structures (that were originally introduced to increase performance) are actually the reason that this algorithm appears to be slightly slower than the other 3 (more modern) algorithms. It is interesting to note that FRM-based NRM is closely clustered to ODM, SDM and LDM, as these 3 algorithms are based on DM. The underlying algorithmic feature that unifies all 4 algorithms is a Reaction Dependency Graph (RDG), which is therefore responsible for the similarly improved performance profiles attributed to these algorithms.

DM has a surprisingly strong performance profile when compared to more modern algorithms, suggesting that subsequent performance improvements introduced actually have little impact for many models. However, Region B in Figure 5.8 highlights a set of BioModels where DM performs quite poorly in regard to algorithms such as ODM and SDM. Region B contains the cluster of the largest reaction networks in the dataset, indicating that DM scales poorly with reaction network size. One should also note that DM does not appear to strongly outperform SDM, ODM and LDM for

any models. This result does indicate that an algorithm such as SDM can completely replace the use of DM for any model.

As shown in Figure 5.6, PDM is the algorithm that is fastest for the largest proportion of BioModels that were benchmarked. Figure 5.8 confirms that PDM is indeed one of the fastest algorithms for most models. However, Region C in the heat map demonstrates that there are models where PDM is actually outperformed by the original DM algorithm it is based upon. Region C contains a cluster of small reaction networks that are differentiated by having a higher number of species than reactions compared to other models. PDM scales with the number of species (other algorithms typically scale with number of reactions) which explains why PDM has relatively poor performance for these models. PDM uses “partial propensities” (see Section 2.4.8) and an advanced species dependency graph data structure to boost computational performance. It is the only algorithm evaluated that uses a species dependency graph, and its optimisations mean that the algorithm claims to scale with species rather than reactions. Whilst PDM’s optimisations give it a strong advantage over other algorithms for most models, the fact it can be outperformed by the original DM algorithm for certain models is surprising. With this type of performance profile, PDM joins TL as an algorithm that would be an important addition to a modern simulation suite.

5.6 Quantitative BioModels performance analysis

As described in Section 5.4, *rps* performance values for 10 runs were collected for each BioModels model-algorithm combination. The Shapiro-Wilk test (see Appendix A.4) was used to determine whether algorithm performance was distributed normally across the runs. It was found that performance across runs was not normal for 49.27% of model/algorithm combinations ($p\text{-value} \leq 0.05$). The Shapiro-Wilk test also showed that mean performance across all models is not normal for any algorithm ($p\text{-value} \leq 0.05$, $\max. W = 0.074$). Therefore, I used non-parametric statistical tests to measure the significance of performance differences between the algorithms.

The distributions of algorithm performance were compared using Kruskal-Wallis H test (see Appendix A.3). For each model in the BioModels dataset, I tested the hypothesis that the performance values of all algorithms come from the same distribution. This hypothesis was rejected for all 380 models ($p\text{-value} \leq 0.01$, $H = 1552.01$).

cr	dm	nrm	frm	ldm	sdm	tl	odm	pdm
0	1	1	2	9	43	75	87	162
0.00%	0.26%	0.26%	0.53%	2.37%	11.32%	19.74%	22.9%	42.63%

TABLE 5.3: Distribution of best performing algorithms for all models in the BioModels dataset. This data is represented as a histogram in Figure 5.6.

Table 5.3 shows the number of times an algorithm was considered the fastest for each model in the BioModels dataset. As discussed in the qualitative analysis based on Figure 5.8 (see Section 5.5), this metric underrepresents the variation in performance of a particular algorithm. To corroborate the findings in Section 5.5, I needed to perform statistical tests that quantify a ranking of algorithms using the performance data generated for all models.

A Mann Whitney U test (see Appendix A.2) was used to perform a pairwise comparison of algorithm performance, with the Benjamini-Hochberg procedure applied to control FDR (false discovery rate) with multiple comparisons. Mean algorithm performance was calculated from all runs on a single model, and the distribution of these means across all models was compared for each possible algorithm pair. For each algorithm pair that had a significant difference in distributions ($p\text{-value} \leq 0.05$), a Spearman’s rho statistic (see Appendix A.5) was used to determine the best performing algorithm. Algorithms were then ranked by the number of other algorithms they outperformed, and in the case of ties the ranks were averaged (see Table 5.4).

The analysis presented in Table 5.4 reveals an interesting result regarding ODM and LDM. LDM is tied in ranking with ODM yet the performance profile of ODM appears stronger in Figure 5.8. Table 5.3 shows that ODM is the best performing algorithm for 87 models compared to only 9 models for LDM, therefore I had previously expected ODM to be ranked more highly than LDM.

pdm	sdm	odm & ldm	tl & nrm	dm	cr	frm
1.0	2.0	3.5	5.5	7.0	8.0	9.0

TABLE 5.4: Algorithm ranking results from Mann Whitney U test using a pairwise comparison of mean algorithm performance for each model. Where the distributions were significantly different ($p\text{-value} \leq 0.05$), a rho statistic was used to determine the best performing algorithm. Algorithms were ranked by the number of other algorithms they outperformed. In the case of ties, the ranks were averaged.

Furthermore, the bi-cluster dendrogram in Figure 5.8 indicates that the performance profiles of ODM and SDM are extremely similar, though ODM is the fastest algorithm for 87 models compared to 43 for SDM (see Table 5.3). These preliminary results suggest that the rank for SDM and ODM would be approximately equal with ODM having a slight performance advantage. However, the analysis reported in Table 5.4 actually shows SDM to be ranked higher than ODM.

The general order of rankings for other algorithms shown in Table 5.4 does appear to fit the results shown in Section 5.5. PDM has consistently been the best performing algorithm for the BioModels dataset, whilst FRM and CR are the worst performers.

pdm	odm	sdm	ldm	tl	nrm	dm	cr	frm
2.48	2.52	2.73	3.77	5.54	5.55	5.74	7.88	8.77

TABLE 5.5: Algorithm ranking results from Mann Whitney U test using a pairwise comparison of algorithm performance based on the distribution of values for all runs for each model. Final algorithm rank was determined by averaging the algorithm ranks across all models.

A further analysis was performed using the pairwise Mann Whitney U test outlined previously to compare algorithm performance, but with some modifications. Unlike the previous analysis (Table 5.4) which used mean performance values, this next analysis used the distribution of values of all runs for each model. In this analysis, algorithm rank was determined by averaging the rank of a particular algorithm across all models (see results in Table 5.5). One major advantage of this analysis over the previous Mann Whitney U test is that information is not lost by considering the performance data of all runs as opposed to using mean values. Another benefit is

that the new ranking now incorporates the relative performance difference between algorithms.

In Table 5.5, one can observe striking differences to the previous analysis (see Table 5.4) with regards to the order of ODM, SDM and LDM. The updated ranking order now corroborates the findings of my preliminary analysis (see Section 5.5). The relative ranking difference between PDM (2.48) and the second highest ranked algorithm, ODM (2.52), is minor. One can also see that the relative rankings of ODM (2.52) versus SDM (2.73) are quite close which fits the clustering of their performance profiles in Figure 5.8, but is not reflected in Table 5.3. The average ranking for DM (5.74) is similar to the average rankings of TL (5.54) and NRM (5.55), demonstrating its relatively strong performance compared to more advanced algorithms. The average rankings of CR (7.88) and FRM (8.77) are much lower than for the other algorithms, which is reflected in the previous performance analyses.

5.7 Summary & conclusions

This chapter has introduced the SSA performance benchmarking suite that I have developed for this thesis. The benchmarking suite allows developers to check algorithm accuracy, assess SSA performance and generate model metric data (see Chapter 4). My performance benchmarking of the BioModels dataset along with quantitative analysis of results has comprehensively tested the first hypothesis of the thesis: *There is no single SSA that is superior in performance for every biomodel*. A simple winners ranking (see Figure 5.6) shows that whilst PDM is the most frequent winner for the BioModels, there are 3 other algorithms that are superior for large numbers of models: ODM, SDM and TL. Statistical testing to determine algorithm ranking based on per model algorithm performance (see Table 5.5) has shown that whilst PDM is the highest ranked algorithm, ODM and SDM's overall rank rating is extremely similar to the PDM value.

The finding that no single algorithm is superior over all models is highlighted by the bi-cluster analysis of model-algorithm performance (see Figure 5.8). Furthermore,

the bi-cluster analysis reveals that one should not simply select one of the highest ranking SSAs from the statistical analysis and expect good performance over *all* models. One can see that for models there are orders-of-magnitude performance differences between high ranking SSAs (see Figure 5.7). In addition, high ranking algorithms each have groups of models that have better performance with other SSAs, including much lower ranked SSAs. This is clear evidence that a scientist should not apply a single SSA formulation to every biochemical model they wish to execute. Instead, a different approach must be adopted as suggested by the third hypothesis of the thesis: *An algorithm can select the best SSA for any arbitrary model with only a small margin of error.*

Now that both model topological metrics and model-algorithm performance data has been collected for each of the BioModels, it is possible to test the third hypothesis. According to the second hypothesis of the thesis, there is a relationship between model characteristics and algorithm performance. Therefore, if the second and third hypotheses hold, it would be possible to predict the fastest SSA for a given model if one is able to extract model characteristics *a priori* to simulation.

Chapter 6

Principled Selection of Stochastic Simulation Algorithm

6.1 Introduction

The availability of multiple variants of the SSA comes at the cost of a lack of clarity as to which one to use for a particular biochemical model. More specifically, many published SSAs are tested with an insufficient number of models, mostly tailored to properties of the newly introduced algorithm. Consequently, it is hard to extrapolate or compare performance between algorithms as each will often be benchmarked against competitors' algorithms using only biochemical models that perform favourably with the newly introduced algorithm. When considering these variants, a scientist may wish to know which SSA will be the fastest for simulating their particular model(s).

Currently, it is common to execute reaction networks with a single SSA implementation, for example Next Reaction Method (NRM) [25]. Due to the lack of model and algorithmic analysis available, scientists are unaware that a different algorithm may perform orders of magnitude faster than their “default” algorithm. To compound this issue, whilst several stochastic simulators are freely available [101, 102, 106, 107],

their selection of algorithms is limited. Such a situation would result in a scientist limiting the complexity of their model to obtain a tractable simulation time. Therefore, it is preferable that scientists are provided with tools that match the best algorithm for their model and allow for better performing simulations. If simulation time can remain tractable in spite of increasing model complexity, the development of finer grained biological knowledge is possible.

The cost of simulating a system with one SSA variant or another depends on the properties of the underlying network of the model and the states reached during the simulation. Each biological model exhibits characteristics that may be suited to a particular simulation algorithm. Thus effective discrimination between SSAs should be based on matching model characteristics to algorithmic performance. In this chapter, I investigate the possibility of using the data generated by the model property analysis (see Chapter 4) and algorithm-model performance benchmark analysis (see Chapter 5) to predict the fastest algorithm for an arbitrary model.

6.2 Experimental roadmap

Thus far, 2 datasets have been benchmarked and analysed: the 8 curated models (see Section 3.2.1) and the 380 BioModels (see Section 4.2.2). I now wish to test whether one is able to predict relative algorithm performance using model characteristics based on the previous benchmark analysis (see Chapter 5). The BioModels use fixed values for species amounts and stochastic rate constants, whilst the curated models are representative of “real world” stochastic models. I have shown how to extract quantitative model characteristics using graph topological analysis of biochemical networks (see Chapter 4).

An initial experiment to detect a relationship between model characteristics and algorithm performance was conducted using linear regression to predict algorithm performance (see Section 6.3). The positive results of this experiment have led me to hypothesise that one can predict the fastest algorithm for an arbitrary model given the model topological properties. Furthermore, statistical testing of the characteristic

equivalence between the 2 datasets indicated that there are many properties that share similar distributions (see Section 4.4.3). Therefore, I hypothesise that it would be possible to use the larger BioModels dataset as a training set for prediction and still obtain accurate prediction for the curated models. A set of prediction experiments were designed to verify these hypotheses.

The prediction experiments are divided into 2 distinct stages. The first stage, (***cross-validation experiments***), focused on the BioModels dataset. Employing 10-fold cross-validation (see Appendix B.5), I wished to determine the quality of predictors that could be generated from this dataset. The second experimental stage (***real world model experiments***) tested the accuracy of predictors generated from the BioModels dataset when used to predict the algorithm performance rankings of the curated models (which were not used in the training set).

In both experimental stages I used four classification methods (see Section 6.5). For a baseline comparison I have used two random predictors (blind pick and distribution aware). The accuracy of each predictor is tested with 4 *relaxation thresholds* ($\varepsilon = [0\%, 1\%, 5\%, 10\%]$). In this scenario, if the performance of the algorithm selected by a predictor for a given model lies within the relaxation threshold of the performance of the *actual best* algorithm, the prediction is scored as correct. In practice choosing an algorithm that performs very closely to the absolute best algorithm would be good scenario for a potential user. The small performance difference is acceptable as long as one is able to avoid the worst performing algorithms. Predictions are scored as correct as long as the performance of the predicted algorithm P_p is worse by no more than ε percentage of the best algorithm performance P_b , that is $P_b - P_p \leq P_b \varepsilon \%$.

Presented with a large number of models properties, I applied some feature selection techniques (see Section 6.4) to determine properties that would potentially improve the prediction accuracy of algorithm performance classifiers. Another set of important properties that would be explored experimentally were properties that were *fast* to compute (highlighted by [†] in Table 4.2) as these would be the strongest candidates to be used in a tool that could predict algorithm performance of a particular model *a priori* to simulation.

For each of the 2 experimental stages, I used 4 different sets of model properties. The first set of properties were those identified by multiple *feature selection* techniques (listed in Table 6.1). Secondly, I tested the performance of classifiers with the *intersection of the set of feature selected properties and the set of properties that were fast to compute*. The third set of properties tested was the *full set of fast properties*. The final experiment of each experimental stage tested classifier accuracy using the *entire set* of available properties. Section 6.6 shows the results of the prediction experiments in comprehensive detail.

Following the prediction experiment results, I explored the impact of classifier mis-predictions (see Section 6.7). The final performance prediction experiment evaluates the accuracy of the best predictors when confronted with much larger models than were available in the BioModels training set (see Section 6.8).

6.3 SSA performance estimation using linear regression

Linear regression (ordinary least squares method) was used to fit a linear model estimating the performance (see Appendix B.1). Regression was performed on a per algorithm basis with results visualised in Figure 6.1. The red data points in each plot show the actual algorithm performance for every BioModel; BioModels are sorted on the x-axis by algorithm performance. The grey data points show the performance estimated by the linear regression model for each algorithm. The coefficient of determination (R^2) is close to 1 (perfect fit) for 7 out of 9 algorithms. PDM and TL algorithms are the exception with R^2 of 0.71 and 0.6 respectively, which indicates their performance is more difficult to estimate with a linear function.

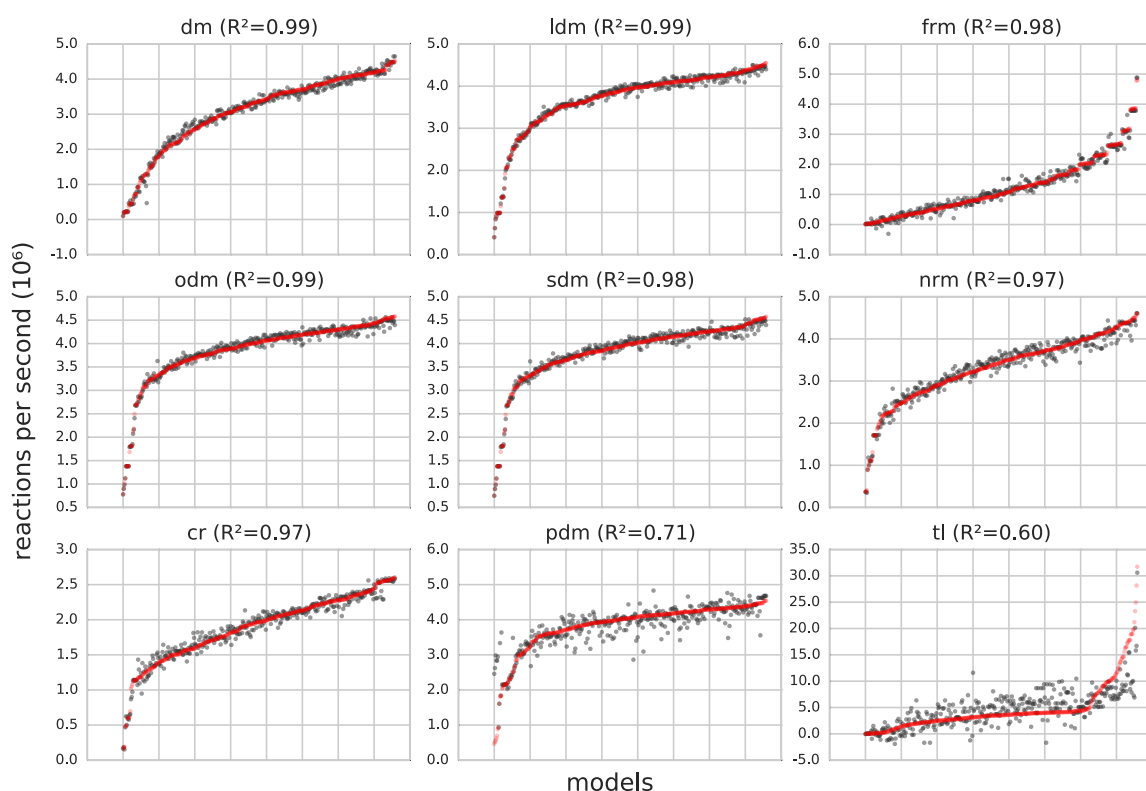


FIGURE 6.1: Comparison of real and estimated performance for each algorithm and all BioModels using linear regression. The red data points are the real algorithm performance values for each model. The grey data points are the performance values estimated by linear regression. The linear regression fit was measured with the coefficient of determination (R^2).

6.4 Feature selection for SSA performance estimation

Linear regression was used to construct a linear model of performance for each algorithm. The absolute value of the coefficients assigned to each property was used for selection. Cross-correlation was calculated between all properties and performance values and then converted to an *F-score* to obtain a p-value for each property. Then properties with the lowest p-values were selected. The last selection method used support vector regression (SVR) [108, 109] with recursive feature elimination (RFE) as an estimator of the performance. The estimator was trained on initial set of all properties and the properties with the smallest absolute weights assigned to them were removed from the set. The procedure was recursively repeated on the reduced set until the desired number of properties was left in the set.

property name	frequency		
	LR	CC	RFE
(SDG) min. directed edge betweenness	7	9	8
(SDG) edge connectivity	3	7	9
(RDG) min. directed betweenness	5	8	6
(SDG) min. out degree	4	7	7
(SDG) min. total degree	4	3	8
(SDG) min. in degree	4	1	9
(RDG) edge connectivity	2	3	7
(SDG) reciprocity	1	2	7
(SDG) min. undirected edge betweenness	5	2	2
(SDG) transitivity	1	5	2
(RDG) max. incoming closeness	1	4	2
(RDG) average local transitivity	2	3	2
(RDG) mean undirected closeness	1	4	1
(RDG) mean incoming closeness	2	2	1
(SDG) max. undirected closeness	1	1	1
(SDG) min. directed betweenness	5	7	-
(RDG) min. in degree	-	2	7
(RDG) max. undirected closeness	-	5	3
(SDG) min. undirected betweenness	2	5	-
(RDG) transitivity	1	-	2
(RDG) max. outgoing closeness	1	2	-

TABLE 6.1: Frequency of properties selection for each of the selection methods: linear regression (LR), cross-correlation (CC) and recursive feature elimination (RFE).

Only properties selected by at least two methods are shown.

Each method selected the top 10% of properties for each algorithm. I have listed the properties most frequently selected across the algorithms for each selection method in Table 6.1. There were three properties selected for all the algorithms — *directed edge betweenness (SDG)*, *edge connectivity (SDG)* and *minimum in degree (SDG)*. It is interesting to note that these three properties are based on the species dependency graph although these systems are typically thought of as reaction networks. *Out*, *in* and *total degree* of the species dependency graph are commonly selected features, which is a strong indication that the density of the species dependency graph influences algorithm performance. Amongst the most highly selected properties there is also *edge connectivity*, *minimum edge betweenness centrality (directed and undirected)* and *reciprocity* for the species dependency graph. These properties all measure the amount of possible traffic passing through the graph, and thus I suspect that bottlenecks in the species dependency graph are likely to be a cause of poor algorithmic

performance. When it comes to the reaction dependency graph, *minimum directed betweenness* and *edge connectivity* are high on the feature selection list. However, the selection of reaction dependency graph based properties is not particularly uniform across the selection methods.

6.5 SSA performance classifier experiments: Methods

I employed two variants of a random predictor, a classifier based on a set of linear regression estimators trained separately for each algorithm (see Appendix B.1), logistic regression [110] (see Appendix B.3), support vector classifier with linear kernel [111] (see Appendix B.2) and a nearest neighbour classifier [112] using a vote of 5 nearest models (see Appendix B.4). For each predictor I performed a 10-fold cross-validation experiment (see Appendix B.5) and measured the mean accuracy and standard deviation.

The two random predictors used different amounts of information. First was a blind random predictor, which assumed that each algorithm is equally probable to perform best. The probability of a such blind guess to be correct is equal to $\frac{1}{9}$ (see Equation 6.1).

$$p = \sum_i w_i p_i = \frac{1}{9} \sum_i p_i = \frac{1}{9} \quad (6.1)$$

The second random predictor assumed that each algorithm is as probable for selection as it was observed as a winner for the training set (see Table 5.3), then roulette wheel selection was used to make a prediction. In the ideal case, this informed random predictor would assign a weight equal to the true probability of winning for each algorithm (see Equation 6.2). Given the distribution of winners in my benchmark I expect the probability of a correct guess to be almost three times greater than in the case of the blind guess.

$$p = \sum_i w_i p_i = \sum_i p_i^2 \simeq 0.27 \quad (6.2)$$

6.6 SSA performance classifier experiments: Results

6.6.1 Cross-validation experiments

To evaluate the accuracy of classifiers, I employed 10-fold cross-validation (see Appendix B.5). Models were split into 10 different folds, that is pairs of training and test sets. Each test set was unique and contained 10% of the models. The order of models in the sets was randomly shuffled but a fixed random seed was used to ensure that all predictors are trained and tested with the same sets.

The first cross-validation experiment investigated the accuracy of predictors using properties identified by feature selection techniques which are shown in Table 6.1. Results for this first experiment are shown in Table 6.2.

predictor	accuracy			
	$\varepsilon = 0\%$	$\varepsilon = 1\%$	$\varepsilon = 5\%$	$\varepsilon = 10\%$
random (blind)	0.12 ± 0.018	0.17 ± 0.021	0.32 ± 0.040	0.42 ± 0.055
random (informed)	0.32 ± 0.024	0.39 ± 0.048	0.54 ± 0.031	0.67 ± 0.035
linear regression	0.32 ± 0.066	0.35 ± 0.061	0.46 ± 0.074	0.58 ± 0.066
logistic regression	0.47 ± 0.074	0.56 ± 0.085	0.67 ± 0.052	0.76 ± 0.065
k-NN vote (k=5)	0.60 ± 0.055	0.67 ± 0.043	0.75 ± 0.053	0.81 ± 0.030
linear SVC	0.50 ± 0.066	0.57 ± 0.054	0.67 ± 0.044	0.76 ± 0.057

TABLE 6.2: Results of the 10-fold cross-validation classification experiment using a reduced set of properties identified in feature selection (see Table 6.1) with increasing relaxation threshold ε .

The experimental accuracy of the random predictors is in agreement with the expectations from theory (see Section 6.5). The informed (distribution aware) random selection is three times more accurate than blind random selection. As expected, the greater the relaxation threshold ε , the greater the classifier accuracy as the set of winners gets larger and it becomes easier to select a winner. Interestingly, accepting as

little as 1% performance difference leads to as much as 9% improvement in selection accuracy. One can also observe that the best predictor accuracy ratio (to the blind guess) drops with increasing ε reaching 2:1 for $\varepsilon = 10\%$. All 4 predictors demonstrate accuracies far higher than a blind random selection, but one must note that linear regression fares worse over all relaxation thresholds than informed random selection. Of the other 3 predictors, k-Nearest Neighbour is the most promising predictor with 60% accuracy (81% with $\varepsilon = 10\%$), compared to blind random selection accuracy of 12% (42% with $\varepsilon = 10\%$).

predictor	accuracy			
	$\varepsilon = 0\%$	$\varepsilon = 1\%$	$\varepsilon = 5\%$	$\varepsilon = 10\%$
random (blind)	0.12 ± 0.018	0.17 ± 0.021	0.32 ± 0.040	0.42 ± 0.055
random (informed)	0.32 ± 0.024	0.39 ± 0.048	0.54 ± 0.031	0.67 ± 0.035
linear regression	0.36 ± 0.066	0.44 ± 0.058	0.56 ± 0.077	0.68 ± 0.061
logistic regression	0.42 ± 0.060	0.50 ± 0.057	0.63 ± 0.061	0.74 ± 0.068
k-NN vote (k=5)	0.46 ± 0.079	0.54 ± 0.072	0.66 ± 0.064	0.74 ± 0.055
linear SVC	0.42 ± 0.056	0.50 ± 0.050	0.62 ± 0.050	0.74 ± 0.059

TABLE 6.3: Results of the 10-fold cross-validation classification experiment using the intersection of the set of properties where (computational complexity $\leq O(V + E)$) and the set of properties identified by feature selection, with increasing relaxation threshold ε .

Although the above results are encouraging, the use of many of the available topological properties is not practical. A large number of these properties have high computational complexity (with respect to number of nodes in the graph), with quadratic complexity for the *betweenness* or *shortest path* measures, up to the order of $O(n^5)$ for *connectivity*. Therefore, for the larger models analysed, the computation of all the properties is more time consuming than running the simulation with all algorithms. In the second experiment, I removed all properties with computational complexity greater than $O(n)$ from the set of feature selected properties (see Table 6.3). Compared to the previous experiment there was a small but noticeable overall drop in prediction quality. k-Nearest Neighbour was still the best predictor but accuracy had dropped from 60% to 46% (81% to 74% with $\varepsilon = 10\%$). Whilst all other predictors had decreased in accuracy, linear regression actually increased from 32% to 36% (58% to 68% with $\varepsilon = 10\%$).

predictor	accuracy			
	$\varepsilon = 0\%$	$\varepsilon = 1\%$	$\varepsilon = 5\%$	$\varepsilon = 10\%$
random (blind)	0.12 ± 0.018	0.17 ± 0.021	0.32 ± 0.040	0.42 ± 0.055
random (informed)	0.32 ± 0.024	0.39 ± 0.048	0.54 ± 0.031	0.67 ± 0.035
linear regression	0.42 ± 0.075	0.47 ± 0.077	0.56 ± 0.097	0.70 ± 0.067
logistic regression	0.60 ± 0.086	0.68 ± 0.063	0.76 ± 0.052	0.82 ± 0.044
k-NN vote (k=5)	0.63 ± 0.070	0.73 ± 0.054	0.80 ± 0.034	0.84 ± 0.046
linear SVC	0.63 ± 0.078	0.72 ± 0.061	0.81 ± 0.060	0.85 ± 0.055

TABLE 6.4: Results of the 10-fold cross-validation classification experiment using a reduced set of properties (computational complexity $\leq O(V + E)$) with increasing relaxation threshold ε .

For the third experiment, I decided to assess the performance of predictors when given the full set of 32 computationally inexpensive (*fast*) properties. Results displayed in Table 6.4 indicate a marked improvement in accuracy over the previous experiments. k-Nearest Neighbour and linear SVC now had the same prediction accuracy (63% with $\varepsilon = 0\%$). However, from evaluating the prediction quality at differing relaxation thresholds, it appears that linear SVC was the strongest predictor (85% with $\varepsilon = 10\%$).

predictor	accuracy			
	$\varepsilon = 0\%$	$\varepsilon = 1\%$	$\varepsilon = 5\%$	$\varepsilon = 10\%$
random (blind)	0.12 ± 0.018	0.17 ± 0.021	0.32 ± 0.040	0.42 ± 0.055
random (informed)	0.32 ± 0.024	0.39 ± 0.048	0.54 ± 0.031	0.67 ± 0.035
linear regression	0.42 ± 0.082	0.48 ± 0.090	0.59 ± 0.098	0.69 ± 0.085
logistic regression	0.64 ± 0.075	0.73 ± 0.052	0.81 ± 0.056	0.86 ± 0.050
k-NN vote (k=5)	0.64 ± 0.075	0.72 ± 0.070	0.80 ± 0.050	0.86 ± 0.041
linear SVC	0.65 ± 0.085	0.75 ± 0.041	0.82 ± 0.035	0.86 ± 0.035

TABLE 6.5: Results of the 10-fold cross-validation classification experiment using the total set of available properties with increasing relaxation threshold ε .

For the final experiment of this stage, I tested the prediction accuracy with the entire set of 100 properties, results are shown in Table 6.5. Prediction accuracy overall was similar to the previous (fast properties) experiment but demonstrated some slight improvement. Linear SVC was still the strongest predictor and has improved from 63% to 65% (85% to 86% with $\varepsilon = 10\%$). This result has highlighted the trend of higher quality predictions when more properties are introduced in the cross-validation experiments (effectively testing on the training set). However, it is important to note

that just using computationally inexpensive properties produced similar quality results to using the full set of properties. It is also surprising that feature selected properties were much less accurate for the classifiers than this fast property set. This may be due to the differences between the feature selection techniques and the classification methods employed. However, linear regression was used in feature selection yet was the worst performing non-random classifier for feature selected properties.

6.6.2 Real world models experiments

In this experimental stage, I decided to determine whether the predictive analysis for algorithmic performance based on the BioModels dataset could be applied successfully to complete “*real world*” models from computational biology literature (Table 3.1). Therefore, I repeated the 4 previous experiments but instead used the curated models as the test dataset. To emulate a black box prediction scenario, I did *not* normalise the values of properties across the models.

predictor	accuracy			
	$\varepsilon = 0\%$	$\varepsilon = 1\%$	$\varepsilon = 5\%$	$\varepsilon = 10\%$
random (blind)	0.14	0.14	0.14	0.29
random (informed)	0.29	0.29	0.43	0.43
linear regression	0.29	0.29	0.29	0.29
logistic regression	0.43	0.43	0.43	0.43
k-NN vote (k=5)	0.14	0.14	0.14	0.14
linear SVC	0.43	0.43	0.43	0.43

TABLE 6.6: Prediction accuracy of complete curated models using a reduced set of properties identified in feature selection (see Table 6.1) with increasing relaxation threshold ε . The entire BioModels dataset was used for training of predictors.

Table 6.6 shows the results of the predictors with feature selected properties. The first interesting result to note is that there is very little difference in accuracy between relaxation thresholds compared to the cross-validation experiments. k-Nearest Neighbour had been the strongest predictor in the corresponding cross-validation experiment but was now the worst, with a selection accuracy equivalent to a blind

random guess at just 14%. Linear SVC and logistic regression were the strongest predictors with 43% selection accuracy.

predictor	accuracy			
	$\varepsilon = 0\%$	$\varepsilon = 1\%$	$\varepsilon = 5\%$	$\varepsilon = 10\%$
random (blind)	0.14	0.14	0.14	0.29
random (informed)	0.29	0.29	0.43	0.43
linear regression	0.29	0.29	0.43	0.43
logistic regression	0.57	0.57	0.71	0.71
k-NN vote (k=5)	0.43	0.43	0.43	0.43
linear SVC	0.43	0.43	0.57	0.57

TABLE 6.7: Prediction accuracy of complete curated models using the intersection of the set of properties where (computational complexity $\leq O(V + E)$) and the set of properties identified by feature selection, with increasing relaxation threshold ε .

The entire BioModels dataset was used for training of predictors.

In Table 6.7, one can see the results of the intersection of the set of fast properties and the set of feature selected properties. There was a marked improvement in prediction quality for k-Nearest Neighbour which now has 43% selection accuracy. Therefore, unlike the cross-validation experiments which improved as more properties were used for prediction, a situation arose where removing certain properties actually increased prediction quality. Logistic regression is the strongest predictor at an improved 53% (71% with $\varepsilon = 10\%$).

predictor	accuracy			
	$\varepsilon = 0\%$	$\varepsilon = 1\%$	$\varepsilon = 5\%$	$\varepsilon = 10\%$
random (blind)	0.14	0.14	0.14	0.29
random (informed)	0.29	0.29	0.43	0.43
linear regression	0.29	0.29	0.43	0.43
logistic regression	0.71	0.71	0.86	0.86
k-NN vote (k=5)	0.57	0.57	0.57	0.57
linear SVC	0.43	0.43	0.57	0.71

TABLE 6.8: Prediction accuracy of complete curated models using a reduced set of properties (computational complexity $\leq O(V + E)$) with increasing relaxation threshold ε . The entire BioModels dataset was used for training of predictors.

The results of the experiment that used all 32 computationally inexpensive properties are shown in Table 6.8. Prediction accuracy had improved for logistic regression,

k-Nearest Neighbour and linear SVC over the 2 previous experiments. Logistic regression is still the strongest predictor with 71% selection accuracy (86% with $\varepsilon = 10\%$). In the previous experiment, I had noted that removing properties improved performance, but in this case the opposite had occurred. The common factor between both experiments was the increasing and exclusive use of *fast* properties, indicating that these properties are highly applicable to classifiers. Similarly to the cross-validation experiments, the feature selected properties fared poorly for prediction accuracy compared to computationally inexpensive properties.

predictor	accuracy			
	$\varepsilon = 0\%$	$\varepsilon = 1\%$	$\varepsilon = 5\%$	$\varepsilon = 10\%$
random (blind)	0.14	0.14	0.14	0.29
random (informed)	0.29	0.29	0.43	0.43
linear regression	0.14	0.14	0.14	0.14
logistic regression	0.57	0.57	0.71	0.71
k-NN vote (k=5)	0.29	0.29	0.43	0.43
linear SVC	0.71	0.71	0.86	0.86

TABLE 6.9: Prediction accuracy of complete curated models using the total set of available properties with increasing relaxation threshold ε . The entire BioModels dataset was used for training of predictors.

The final experiment of this stage repeated the use of the entire set of 100 properties for comparison (see Table 6.9). This increased the quality of linear SVC to 71% (86% with $\varepsilon = 10\%$), but reduced the accuracy of the other 3 predictors. Linear regression now has prediction accuracy equivalent to blind random selection at 14%, whilst k-Nearest Neighbour has the same prediction quality as informed random selection (29%). Although the best predictor on the full property set has the same accuracy as the best predictor on the fast property set, there is an overall decrease in selection accuracy.

6.6.3 Prediction results summary

These results demonstrate that one can make predictions of SSA performance for an arbitrary model that is significantly higher than a random selection. Table 6.5 shows

that best classifier accuracy is over 5 times better than a blind random selection¹.

There were 4 property set results used for each experimental stage. Surprisingly, I found that the 2 property sets based on feature selected properties had relatively poor prediction accuracy. The other 2 property sets are the *fast-to-compute* properties and the *complete* property set. The complete set of properties are not suitable for *practical* use as it contains many computationally expensive properties. For large models, this would mean that model property analysis exceeds model simulation time, rendering any such analysis redundant. Fortunately, the experimental results have shown that the subset of fast-to-compute properties has accuracy that is close to the complete set.

If one acknowledges the fast-to-compute properties as the best *practical* choice for a predictor, one can continue this pragmatism by allowing *almost fastest* SSA predictions to be considered correct. With a relaxation threshold $\varepsilon = 10\%$, the best classifier linear SVC has 85% accuracy for the cross-validation experiments (see Table 6.4). Using the “*real world*” models test set achieved 86% accuracy versus 29% for a blind random selection, though logistic regression was the best predictor for these models (see Table 6.8). This is clear evidence that using static topological properties provides accurate predictions for fully parametrised (and thus dynamic) stochastic models.

6.7 SSA performance classifier experiments: Assessing mispredictions

The 2 experimental stage results provided different winning classifiers for *fast-to-compute* graph property queries that could be employed in a tool (see Tables 6.4 & 6.8). The cross-validation stage indicated that linear SVC would be the strongest classifier. Using the curated models as the test set suggests that logistic regression should be employed for use in a “*real world*” tool to automate SSA selection. However,

¹Table 6.5 cross-validation experiments show that with no relaxation threshold ($\varepsilon = 0$), a blind random pick has 12% accuracy, whilst the best classifier (linear SVC) has 65% accuracy

one should note that the curated models dataset is small (8 models) and thus may over-represent edge cases. The prediction experiments have *relaxation thresholds* (ε) because for practical uses selecting an SSA with *almost best* performance is sufficient. Crucially, the failures of a tool to automate SSA selection are just as important as its successes. A misprediction that resulted in the selection of an algorithm that was an order of magnitude or more slower than the best would be considered a catastrophic failure for such a tool. Therefore, one should be able to discriminate on the quality of the 2 most highly rated classifiers by assessing the impact of mispredictions.

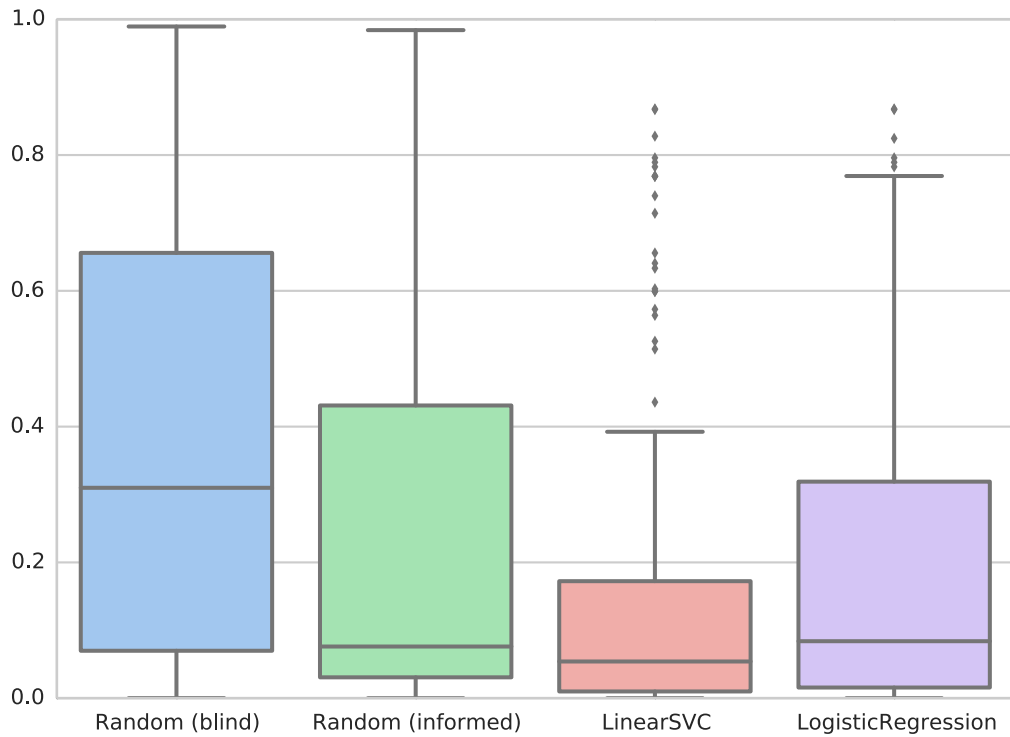


FIGURE 6.2: Distribution of relative performance loss caused by mispredictions for the best (linear SVC & logistic regression) and worst predictors (two variants of random choice). The whiskers represent the 1.5 IQR (interquartile range) past the closest quartile (top/bottom edge of the box). Observations outside this range are marked as outliers.

To demonstrate the consequences of mispredictions for a tool based on our analysis, I ran another 10-fold cross-validation experiment using the fast property subset (computational complexity $\leq O(V + E)$). This time I measured the relative performance loss for each inaccurate prediction. Figure 6.2 shows the distribution of these values

for the best classifiers for both experimental stages (linear SVC and logistic regression) compared to the random predictors. One can see from the box and whisker plots that both classifiers significantly outperform a blind random selection which has median 31% relative performance loss. However, logistic regression (with median 8.4% performance loss) had a higher median performance loss than the distribution aware random selection (median 7.6% loss). On the other hand, logistic regression had a lower interquartile range than the distribution aware random selection.

The predictor with the most favourable mispredictions, linear SVC, had a median relative performance loss value of only 5.4% and the interquartile range was lower than 20%. This means that most of the mispredictions correspond to less than a 20% performance loss, though I also found several outliers for which the performance drop was large (up to 86%).

6.8 SSA performance classifier experiments: Large scale models

The BioModels training set used for the classifiers is mainly composed of smaller reaction network models (see Figure 4.1). One of the aims of this thesis is to develop techniques that allow SSA performance to scale with the growing demands of systems & synthetic biology. Therefore, I designed an experiment to investigate whether the predictors trained on small models would be accurate for large models. I treated model A3 (see Section 3.2.4), as a “template” and instantiated it on 1×1 , 10×10 and 100×100 two dimensional lattices (see Table 6.10 for the model network sizes). Model A3 represents a quorum sensing mechanism within a single *Escherichia coli* cell, and is fully parametrised. Quorum sensing is a simple communication mechanism that occurs between *multiple* cells, therefore it is logical to extend the model in this manner. Transport reactions were added between adjacent cells on the lattice to consider the flow of signalling molecules between cells. Table 6.10 reveals that the

largest model represents a colony of 10,000 cells and consists of 289,000 reactions (the vast majority of BioModels have less than 50 reactions).

model size	reactions	species
1×1	25	21
10×10	2860	2100
100×100	289600	210000

TABLE 6.10: Reaction and species graph sizes for different versions of the stochastic *Escherichia coli* AI-2 quorum signal circuit model from Li et al. The model size is the number of points on a 2D lattice the model was instantiated on.

Figure 6.3 shows the algorithmic performance for the *Escherichia coli* quorum sensing model instantiated for the different lattice sizes. One can observe vastly differing performance profiles between the model variants. The “single cell” version of the model has TL as the clear winner, whilst ODM was the second fastest algorithm (but has less than 50% of TL performance). For 10×10 lattice version, CR was the fastest algorithm closely followed by PDM. CR was also the fastest algorithm for the 100×100 model, but this time by a wide margin. PDM was the second fastest algorithm for this model but was still over 5 times slower than CR. All other algorithms had extremely poor performance for this large model.

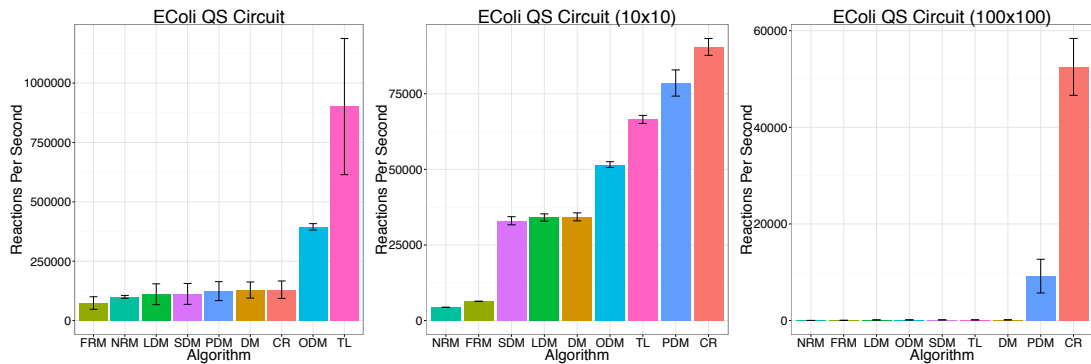


FIGURE 6.3: Algorithmic performance (number of reactions per second) for an *Escherichia coli* quorum sensing circuit. Models were instantiated on a square lattice with single, 100 and 10 000 cells. Transports reactions were added for adjacent cells of the lattice.

To complete the experiment, I used the 2 best classifiers using *fast-to-compute* model properties from the prediction experiments to assess the accuracy with the large

model variants. The logistic regression predictor selected ODM for all 3 model variants. However, the benchmark shows that ODM is significantly slower than the fastest algorithm for all models. ODM ranks second in the single cell model version, which may indicate that prediction quality is higher for smaller models. The linear SVC predictor, selected TL for the single model and PDM for the 10×10 and 100×100 lattices. Whilst the TL prediction is accurate, CR is the fastest algorithm for the large lattices (PDM is second fastest). One can argue that for the 10×10 lattice, a pragmatic relaxation threshold ε should mean that PDM selection is also an accurate prediction. The PDM algorithm is 13% slower than CR for the 10×10 lattice and 82% slower for 100×100 . The BioModels training set has no models for which CR is the winning algorithm, therefore it is impossible for any of the classifiers to predict this result. If one was to disregard CR (as there are no training examples), the linear SVC predictor would have made perfect predictions of the fastest SSA for these models.

6.9 Summary & conclusions

The third hypothesis of this thesis asserted that one should be able to use an algorithm to determine the selection of the best SSA for an arbitrary biochemical model with minimal error. This chapter has presented the use of statistical classification techniques to make an automated selection of the fastest SSA. Cross-validation experiments show selection accuracy of up to 65% compared to 12% for a blind random selection. Prediction experiments using a small set of fully parametrised models indicate a selection accuracy of up to 71%, compared to 14% for a blind random pick. These results confirm the third hypothesis; it is indeed possible to use an algorithmic method to select the best SSA with an accuracy far higher than a random pick. One should note that this implicitly demonstrates that the second hypothesis (*there is a relationship between biomodel characteristics and SSA performance*) also holds because the statistical classification methods employ this relationship to make these predictions.

To enable a transfer of these findings to a practical use case, it is necessary to only consider the *fast-to-compute* model properties (see Table 4.2). Although this restricts the number of available properties to 32 (from 100), I have found that there is only a very minor difference in selection accuracy. For example, Tables 6.4 & 6.5 show that for the cross-validation experiments the best predictor accuracy only drops from 65% to 63%. It is likely that feature redundancy exists between the subsets of *fast-to-compute* and *slow-to-compute* properties. This finding had been predicted by the BioModel property correlation analysis (see Section 4.4.2). Furthermore, if I consider *almost fastest* algorithms (10% relaxation threshold) as optimal selections, up to 86% accuracy is achieved with *fast-to-compute* properties (see Tables 6.4 & 6.8).

I have considered the impact of mispredictions on relative performance loss (see Section 6.7) and found that linear SVC resulted in the lowest performance loss (out of the 2 best performing predictors when using *fast* properties). Linear SVC was also found to be the most accurate predictor for a large scale biosystem example using *fast-to-compute* properties (see Section 6.8) and cross-validation experiments (see Section 6.6.1). Although logistic regression is the best predictor for the fully parametrised curated models, I recommend linear SVC as the classifier for a tool. This is because of the low number of models available in the curated models dataset (see Section 6.6.2), and the relative success of linear SVC for the other experiments and analysis. These results mean that it is feasible to produce a computational tool to improve SSA performance by selecting the fastest SSA based on model characteristics. I subsequently created a web application for this purpose named ***ssapredict*** which is described in the following chapter.

The analysis presented in Chapter 6 was explicitly designed to test the hypotheses of this thesis. As such, the analytical techniques employed were intentionally elementary. For example, the classifiers evaluated were principally based on a linear model in spite of findings indicating that non-linear techniques may provide increased classifier accuracy (see Section 6.3). However, by demonstrating that relationships can be found with relatively simple methods, I am providing transparent evidence that the hypotheses hold. Future work in this area should expand upon this analysis

using more rigorous techniques to discover intricate relationships. An unexpected finding was that classifiers significantly decreased in accuracy with a consensus vote of three simple feature selection methods I had employed: *LR*, *CC* & *RFE-SVR* (see Section 6.4). Therefore, future work should identify important biomodel features by exploring more advanced and/or appropriate feature selection methods. For example, the Correlation Feature Selection (CFS) algorithm is suitable for optimising classifiers as it is designed to generate feature subsets that are “highly correlated with the class, yet uncorrelated with each other” [113].

Furthermore, one can improve the assessment of classifier accuracy by using multiple cycles of cross-validation [114]. Krstajic et al. demonstrate experimentally that a single pass of cross-validation is subject to high variance and thus does not provide accurate assessment. Repeated cross-validation involves psuedo-random generation of folds such that a given fold \mathcal{D}_t (see Appendix B.5) is composed of a different subset of models at each cross-validation cycle. The disadvantage of multiple cross-validation cycles (Krstajic et al. suggest 50 repeats) is an extreme increase in computational time required. However, future work that generates more advanced classifiers should employ this method to avoid errors in the assessment of classifier accuracy.

Another issue to be tackled in future analysis is the present class imbalance problem. Due to the restricted nature of the published models available, the number of positive examples for training were heavily imbalanced by class (see Figure 5.6). For one of the algorithms, *CR*, there were no positive examples and thus this algorithm would never be selected by a classifier. However, the large scale model experiments (see Section 6.8) show that *CR* is actually an algorithm that is applicable to specific types of model. Therefore, it is important to increase the number of positive examples for the underrepresented classes in the dataset to generate more accurate classifiers. Should it prove difficult to curate a balanced dataset in the absence of applicable published models, class imbalance techniques must be adopted. These include resampling strategies that, for example, oversample minority class examples for training [115]. Another approach is to use an ensemble classifier (a combination

of single classifiers) which has been shown to improve classifier performance with imbalanced datasets [116].

Chapter 7

***Ssapredict*: Meta-Stochastic Simulation Web application**

7.1 Introduction

Stochastic simulation algorithms (SSAs) are used to trace realistic trajectories of biochemical systems at low species concentrations. As the complexity of modelled biosystems increases, it is important to select the best performing SSA. Numerous improvements to SSAs were introduced but they each only tend to apply to a certain class of models. This makes it difficult for a systems or synthetic biologist to decide which algorithm to employ when confronted with a new model that requires simulation. In Chapter 6, I demonstrated that it is possible to determine which algorithm is best suited to simulate a particular model, and that this can be predicted *a priori* to algorithm execution. I present a web based tool ***ssapredict*** that allows scientists to upload a biochemical model and obtain a prediction of the best performing SSA [24]. Furthermore, *ssapredict* gives the user the option to download our high performance simulator *ngss* (see Chapter 8) preconfigured to perform the simulation of the queried biochemical model with the predicted fastest algorithm as the simulation engine.

The *ssapredict* web application is available at <http://ssapredict.ico2s.org>. It is free software and its source code is distributed under the terms of the GNU Affero General Public Licence.

7.2 Web application overview

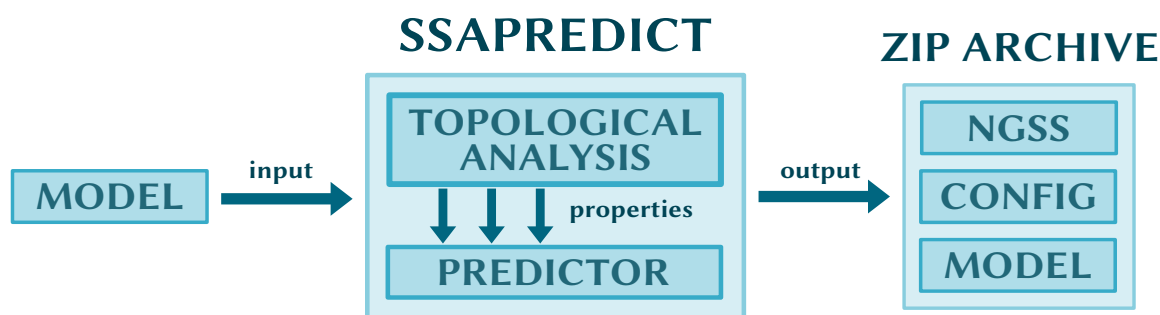


FIGURE 7.1: Structural diagram of *ssapredict* architecture and work-flow [24].

My *meta-stochastic* simulation solution, *ssapredict*, is implemented as a web application. Figure 7.1 provides an overview of the *ssapredict* work-flow. *Ssapredict* is designed to be easy to use and receiving a prediction only requires the user to press an upload button and select the biochemical model of interest (in SBML [33] format) that resides on their computer. After the model has been uploaded, the web application automatically begins model topological analysis using the fast *propertygen* auxiliary application (see Section 7.4). A prediction is then made using the linear SVC classifier that was trained using the BioModels benchmark & property data (see Chapter 6). Once the prediction has been made, the user can download our portable high performance simulator, *ngss* (next generation stochastic simulator) which is preconfigured to run the model with the predicted fastest algorithm (statically built for use with Mac OS X, Linux or Windows operating systems).

7.3 *Ssapredict* walk-through

A user can access the *ssapredict* web application using a modern internet browser by entering the web address: <http://ssapredict.ico2s.org>. Figure 7.2 shows the landing/home page for *ssapredict*.

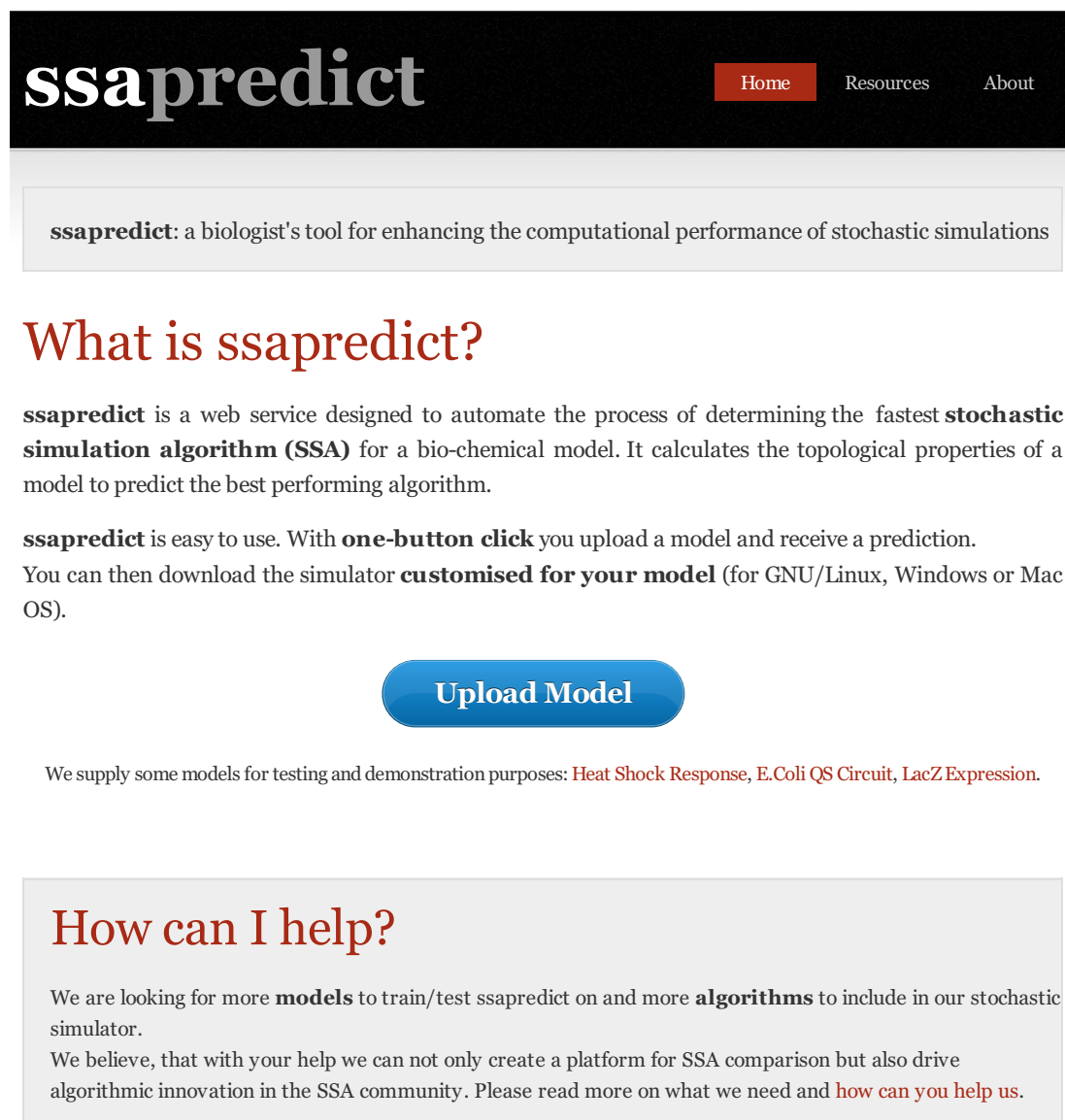


FIGURE 7.2: Screenshot of *ssapredict* “home” page.

At the centre of the page, there is a large button which allows the user to upload a model. The upload button activates a “file open” dialog for a user to select the model from the filesystem. At the top of the page, there is a black header that contains the

name of the application, as well as 3 small buttons. The header persists over multiple pages of the web application, so these buttons are always available but interrupt analysis if used. The *Home* button returns the user to the landing page to perform a new analysis. The *Resources* button takes the user to a page that provides: (1) source code for the web application, (2) source code for the *ngss* simulator, (3) source code for *propertygen* and (4) the 380 BioModels predictor training set. The *About* button lists the members of the *ssapredict* team and gives details to allow SSA developers and modellers to contribute to the meta-stochastic simulation suite. Directly below the upload button there is a selection of SBML models for test purposes (these models have been taken from the curated models dataset).

What is ssapredict?

ssapredict is a web service designed to automate the process of determining the fastest **stochastic simulation algorithm (SSA)** for a bio-chemical model. It calculates the topological properties of a model to predict the best performing algorithm.

ssapredict is easy to use. With **one-button click** you upload a model and receive a prediction.

You can then download the simulator **customised for your model** (for GNU/Linux, Windows or Mac OS).



Get Results

We supply some models for testing and demonstration purposes: **Heat Shock Response**, **E.Coli QS Circuit**, **LacZ Expression**.

FIGURE 7.3: Cropped screenshot of *ssapredict* “home” page after model uploaded.

Figure 7.3 shows a cropped screenshot of the home page after a model has been uploaded. Once the model file is selected, the original upload button is faded out and an orange upload progress bar appears. After the model completes the upload process, a new “Get Results” button is faded on to the page. The results button activates the analysis process and the user is taken to a temporary “progress” page until the analysis completes. In the background, *ssapredict* executes *propertygen* on the uploaded model to extract 32 *fast-to-compute* graph properties. The generated property values are subsequently delivered to the (*BioModels trained*) linear SVC fastest SSA predictor (see Section 7.5).

Upon analysis completion, the prediction results are displayed on the *results* page (see Figure 7.4). For the example *heat shock response* model, *ssapredict* has selected PDM as the fastest SSA. In Section 3.2.3, I performed a SSA performance benchmark (upon *model A2*) which demonstrated that PDM was the actual fastest algorithm for this biochemical system.

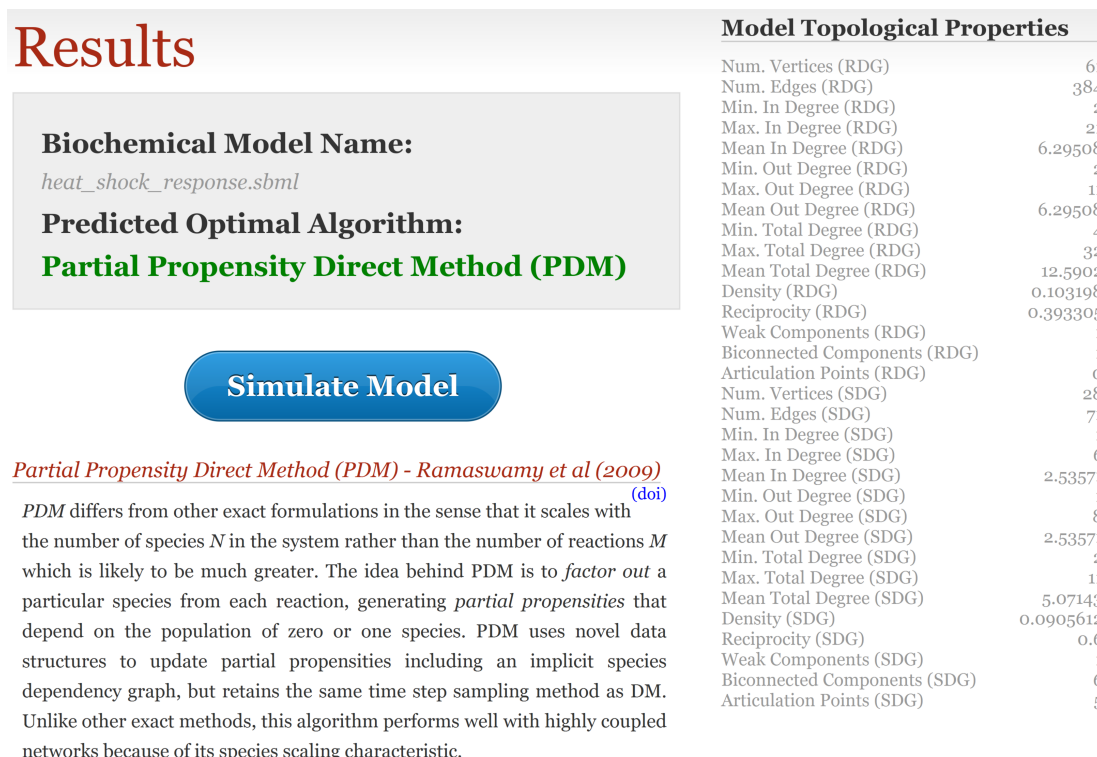
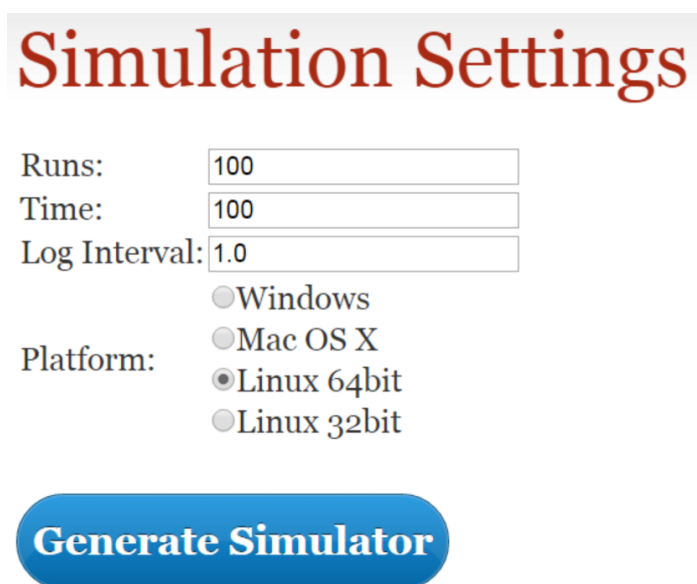


FIGURE 7.4: Cropped screenshot of *ssapredict* results page after model has been analysed. The model *heat_shock_response.sbml* has been analysed and *ssapredict* has selected PDM as the expected fastest algorithm.

The results page also includes a brief summary of the algorithm to explain its optimisations as well as reaction network configurations that it is best suited to. At the right hand side of the page, there is a list of the 32 graph property values computed by *propertygen* should the user wish to perform their own topological analysis. The results page also contains a “*Simulate Model*” button. This feature allows a user to download a static¹ version of the *ngss* stochastic simulator for Mac, Windows or Linux operating systems. The user is taken to a “*Simulation Settings*” web page in order to provide simulation parameters for *ngss* (see Figure 7.5).

¹A static executable in this context is compiled and/or distributed with all dependencies, therefore no libraries need to be installed on a target machine to run the application.



Simulation Settings

Runs: 100

Time: 100

Log Interval: 1.0

Platform:

- ☐ Windows
- ☐ Mac OS X
- ☒ Linux 64bit
- ☐ Linux 32bit

Generate Simulator

FIGURE 7.5: Cropped screenshot of *ssapredict* simulation settings page.

The simulation settings page allows users to provide 4 parameters: (1) number of simulation runs, (2) amount of simulation time to execute the model (in model time units), (3) simulation time intervals at which the simulator logs the system state vector and (4) the target operating system. *Ssapredict* then generates a *ngss* XML simulation parameters file for these values, as well as setting the predicted fastest SSA. *OpenMP* multi-core parallelism and CSV (comma separated values) simulation output settings are also set as default in the parameters file. The web application then assembles a *zip* file (see Figure 7.1) that contains: (1) the *ngss* executable for the target platform, (2) the generated simulation parameters file, (3) the uploaded model file and (4) instructions for using the simulator. The *zip* file format [117] was chosen for compression because it has the widest support over multiple platforms (without installing third party software).

7.4 Model property generation

Uploaded SBML models are parsed and the reaction & species dependency graphs are generated (see Section 4.2). 32 *fast-to-compute* properties are extracted from the RDG and SDG of the model (see Table 7.1). To minimise analysis time and thus

produce a prompt prediction, I had to maximise property generation performance. Therefore, I wrote an auxiliary program in C++ (which has much greater performance than the python language) dedicated to computing graph properties called *propertygen*. *Propertygen* is dynamically initiated by *ssapredict* after model upload, and the output is accessible to the web application. See Section 7.6.2 for technical details.

Computational Complexity	Graph Property
$O(1)$	number of edges, number of vertices, density of graph
$O(V)$	min mean max outgoing edges, min mean max incoming edges, min mean max all edges
$O(V + E)$	weakly connected components, articulation points, bi-connected components, reciprocity of directed graph

TABLE 7.1: Summary of “fast” model topological properties analysed by *propertygen*. Complexity relates to *worst case time complexity* for the computation of the property, where V is vertices, E is edges.

7.5 *Ssapredict* (fastest SSA) predictor

Ssapredict uses the linear SVC (fastest SSA) predictor developed in Chapter 6. The linear SVC predictor is trained with the BioModels dataset (see Section 4.2.2) using SSA performance benchmark results and the subset of 32 *fast-to-compute* properties. Once the *propertygen* model topological analysis has completed, the 32 property values are delivered to the predictor and a prediction of the fastest SSA is returned. See Section 7.6.3 for technical details.

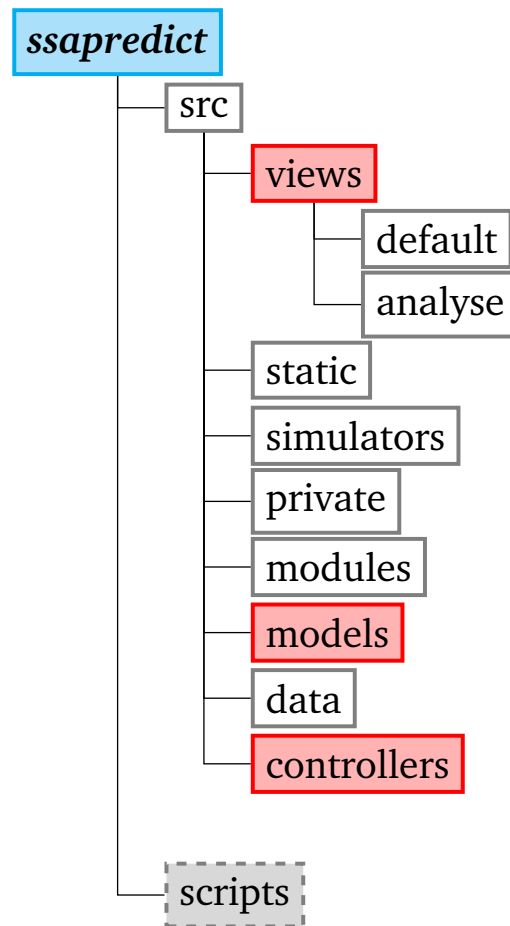
7.6 Software engineering

7.6.1 *Ssapredict* web application

The *ssapredict* web application is based on the python **web2py** web application framework (version 1.99.4) [118, 119]. *Web2py* implements the established *model-view-controller* design pattern which developers must follow to build web applications [120]. The *ssapredict* **model** contains: (1) the menu system (i.e. header buttons), (2) functions that “bootstrap” the prediction analysis tasks and (3) a persistent *SQLite* database of performed analyses [121]. The **view** contains the *HTML* for each web page of the application [122]. The **controller** contains python functions for each web page of the application. The controller dynamically generates content for each of the web pages dependent on application flow, which can then be displayed by the view. Figure 7.6 shows the directory structure of the *ssapredict* web application source code. The *model-view-controller* parts of the web application structure are highlighted in red.

Under the *views* folder, there are *default* and *analyse* sub-folders. The *default* folder contains *HTML* for the *home*, *team* and *resources* pages. The *analyse* folder contains the *analysis progress*, *analysis results* and *simulation settings* pages. The *static* folder contains the aesthetic theme [123] and *javascript/jquery* [124] client-side scripts used by the website. The *modules* folder contains python source code files for functionality that is external to *web2py*: (1) statistical classification, (2) property parsing, and (3) *zip* file generation. The *simulators* folder contains static versions of the *ngss* simulator for Linux, Mac OS X, and Windows operating systems. The *data* folder contains the *linear SVC* fastest SSA predictor. The *private* folder has a *web2py* configuration file that is used for web application server routing. Finally, the *scripts* directory contains functionality for initialising the web application *SQLite* database.

To bootstrap the *web2py* server, the `_scheduler.py` script must first be initialised. The *web2py* scheduler is required for the server to run background tasks; it is required

FIGURE 7.6: Diagram of *ssapredict* file system source tree.

for updating the *SQLite* database. Once the scheduler is initialised, the server can be started using the executable `web2py.py` script.

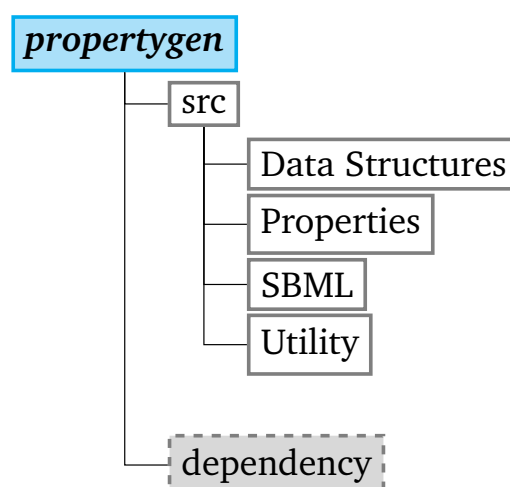
There are two active *controller* files for *ssapredict*: `default.py` and `analyse.py`. Each python function in the controller files is specific to a separate *view*. Values returned by a controller function is accessible to its respective view for display. The `index` controller function uses a `SQLFORM` object and the `jquery.fileupload.js` script to handle model upload. Once the model is uploaded, a new database scheduler_task is created containing a task identifier and model filename. The application is subsequently forwarded to the `analyse` function in `tasks.py` (*models* folder). This function runs *propertygen* on the uploaded model and executes the `predict_model` function from the `prediction.py` module (see Section 7.6.3) on the generated model topological property values to predict the fastest SSA. The

application is subsequently redirected to the `results` controller/view where it displays the predicted fastest SSA and the computed model topological properties data. The “*Simulate Model*” button forwards the user to the `download` controller which uses a `SQLFORM` object to record user inputted simulation settings. The “*Generate Simulator*” button copies the appropriate simulator version and `zip` compresses it with the generated simulation parameters (`zipgen.py` module) and the uploaded model file. A download dialog is initialised for the user, and intermediates are deleted after the download completes.

7.6.2 *Propertygen* SBML model graph property generator

Propertygen is the model topological property analyser auxiliary program used by *ssapredict* to generate 32 *fast-to-compute* properties from the RDG & SDG of a biochemical model (see Chapter 4). *Propertygen* is free software and its source code is distributed under the terms of the GNU General Public Licence (GPL) version 3 [125].

Parsing SBML models, generating dependency graphs and computing graph properties are the most computationally expensive aspects of the prediction workflow (see *topological analysis* in Figure 7.1). Therefore this part of the web application was written in the C++ programming language [126] and compiled to improve performance by orders of magnitude compared to the *python* interpreter employed by *web2py*. Figure 7.6 shows the directory structure of the *propertygen* source code. The *SBML* folder contains the SBML [33] parsing source code that uses the LGPL licensed *libsbml* 5.6.0 library [52]. The *Properties* folder contains source code for graph property generation which is based on the GPL licensed *igraph* 0.5.4 network analysis library [86]. The *Data Structures* folder contains the source code required to generate the RDG and SDG of a biochemical model (see Sections 4.2.4 & 4.2.5). The *Utility* folder provides some simple file output code for writing property results to disk. The *dependency* directory contains a compressed archive of the *igraph* 0.5.4 library, as *propertygen* is incompatible with newer versions of this library.

FIGURE 7.7: Diagram of *propertygen* file system source tree.

The main function of *propertygen* reads a single command line parameter: the SBML filename. The SBML file is then parsed using the *CSBMLReader* class, returning a populated *CModelData* object. The *CPropertiesManager* class invokes generation of the RDG & SDG, and wraps function calls to the *igraph* library. The calculated graph property values are written to a file using the *CFileOutput* class.

7.6.3 Linear SVC model-algorithm performance predictor

Ssapredict uses a Linear SVC predictor trained with the BioModels SSA performance benchmark and model topological properties data (see Chapter 6). This is implemented as a single python module that uses the **scikit-learn** machine learning library [127]. Once *propertygen* has performed a model topological analysis, it writes property values to the file system. *Ssapredict* invokes the `prediction.py` script with the model property values file and a prediction is returned.

Figure 7.8 shows the python function call graph of the prediction module when executed on the property values of model A3 (see Section 3.2.4). The call graph shows that the predictor takes less than a second to run (result from an *Intel Core i5 4200U* laptop). The prediction module first uses `read_data` to access the BioModels training data from the file system. This is the most expensive operation of the prediction module (80% of computational time spent in the `read_data` function).

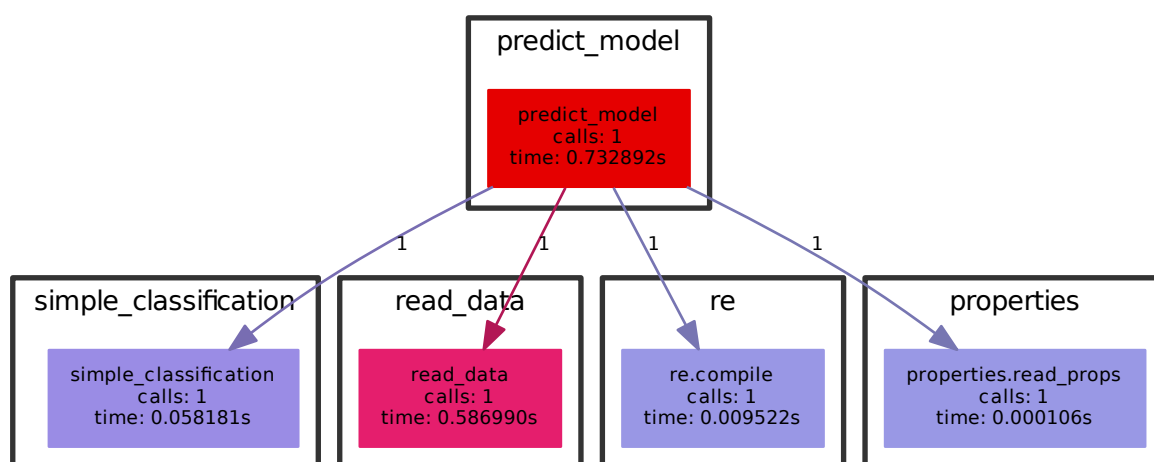


FIGURE 7.8: Function call graph of *ssapredict* python linear SVC prediction module. In this instance, the predictor evaluated the model topology of model A3 (see Section 3.2.4) and predicted ODM as the fastest SSA.

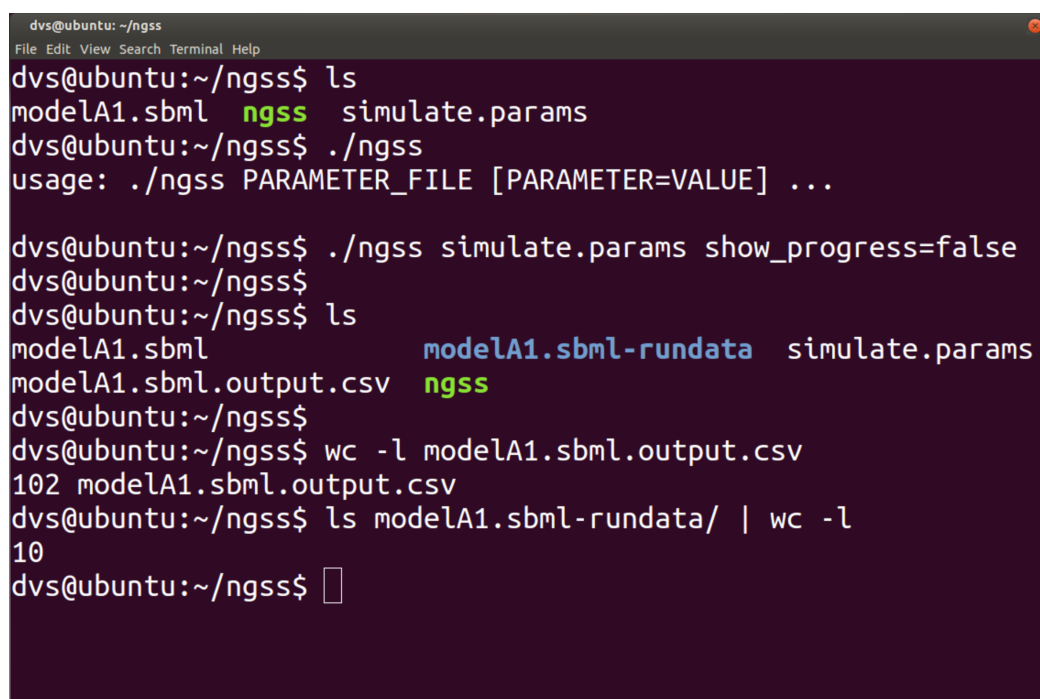
The module then uses regular expressions (*re*) to remove unused data from the training set (1% of CPU time). The 32 model topological property values generated by *propertygen* are read from the file system using *read_props* (<1% of CPU time). One should note that the computational cost of the predictor will not vary by model size, as the predictor always handles the same number of property values at each invocation. Finally, the *simple_classification* function invokes the *sklearn* linear SVC classifier using the training data and target model properties (8% CPU time). Other auxiliary/complementary functions have been removed from the call graph for clarity.

Chapter 8

Next Generation Stochastic Simulator (ngss)

8.1 Introduction

Next generation stochastic simulator (ngss) is the simulation software developed during the course of this thesis. *Ngss* is the natural successor to the *multi-compartment stochastic simulator (mcss)* that was previously developed by colleagues in my research group [22]. The *ngss* SSA implementations (see Table 5.1) are reused from source code originally developed for (and tested by) the SSA benchmarking suite. *Ngss* also retains the computational performance benchmarking capabilities of the SSA benchmarking suite (see Chapter 5). *Ngss* was developed in the C++ programming language with developmental priority assigned to computational performance. The software is written to be cross-platform, and compiles & runs on Mac, Windows and Linux operating systems. *Ngss* is the simulation component of the *ssapredict* [24] SSA prediction & simulation suite (see Figure 7.1).

A terminal window titled 'dvs@ubuntu: ~/ngss' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

```
dvs@ubuntu:~/ngss$ ls
modelA1.sbml  ngss  simulate.params
dvs@ubuntu:~/ngss$ ./ngss
usage: ./ngss PARAMETER_FILE [PARAMETER=VALUE] ...

dvs@ubuntu:~/ngss$ ./ngss simulate.params show_progress=false
dvs@ubuntu:~/ngss$
dvs@ubuntu:~/ngss$ ls
modelA1.sbml          modelA1.sbml-rundata  simulate.params
modelA1.sbml.output.csv  ngss
dvs@ubuntu:~/ngss$
dvs@ubuntu:~/ngss$ wc -l modelA1.sbml.output.csv
102 modelA1.sbml.output.csv
dvs@ubuntu:~/ngss$ ls modelA1.sbml-rundata/ | wc -l
10
dvs@ubuntu:~/ngss$
```

FIGURE 8.1: Terminal window showing example use of the *ngss* simulator.

8.2 Simulating models With *ngss*

Ngss has been designed to run from the *command line interface* (or be executed by another application as an external command). Figure 8.1 shows example use of *ngss* within a Linux terminal window. *Ngss* takes a *XML simulation parameters* file as the mandatory first command line argument. In this example, the parameters file is `simulate.params`. Figure 8.2 displays the contents of the `simulate.params` file used in Figure 8.1. *Ngss* also accepts an arbitrary number of additional command line arguments, each of which override any default (or parameter file) simulation parameter values. In Figure 8.1, the `show_progress` parameter is overridden as *false* on the command line, whilst Figure 8.2 shows that it was originally set to *true* in the `simulate.params` file. The simulation parameters select the `modelA1.sbml` file for simulation and specifies comma separated values (*csv*) for simulation output. Thus, after execution a `csv` file `modelA1.sbml.output.csv` is generated that contains simulation time-series (means and standard deviations) data. A `modelA1.sbml-rundata` directory is also generated that contains the time-series

for each individual run in *csv* format. Figure 8.1 demonstrates using the *UNIX* `wc -l` command to count lines in a file, that there are 100 values (if one ignores the 2 lines with fields and whitespace) in the time-series, and 10 individual-run *csv* files in the `modelA1.sbml-rundata` directory. These values were set in the simulation parameters as 10 runs and 100 simulation `max_time` units, with a `log_interval` of 1.0 time units (see Figure 8.2).

8.3 Simulation parameters

This provides details of the simulation parameters available to configure *ngss* at run-time. This XML parameters system is a clean reimplementation of the *mcss* simulator [21, 22] parameters system (using the RapidXml parsing library [128]). Figure 8.2 shows the structure and fields of an example *ngss* simulation parameters XML file. If one considers the XML document as a tree structure, there is a `parameters` root element which encloses a `parameterSet` element named “*SimulationParameters*”. The `parameterSet` contains a list of `parameter` elements that each have a pair of attributes: *name* of parameter and associated *value*.

Section 8.3.1 lists the parameter *names* along with their *value* types (shown in green). If appropriate, parameter value option strings are also displayed (in grey).

8.3.1 Available simulation parameters & types

model_file (*STRING*) Name of the *sbml* model on the file system. This should be fully qualified with directory relative to the *ngss* executable and include the model file extension.

simulation_algorithm (*STRING*) The stochastic simulation algorithm to use:
“dm”, “frm”, “ldm”, “odm”, “sdm”, “nrm”, “tl”, “pdm”, “cr”.

parser (*STRING*) Parser to use for the model file:
“sbml”, “mcss”.

```

simulate.params + (~/.ngss) - VIM
File Edit View Search Terminal Help
1 <?xml version='1.0' encoding='utf-8'?>
2 <parameters>
3   <parameterSet name="SimulationParameters">
4     <parameter name="parser" value="sbml"/>
5     <parameter name="output" value="csv"/>
6     <parameter name="seed" value="0"/>
7     <parameter name="show_progress" value="true"/>
8     <parameter name="max_runtime" value="0.0"/>
9     <parameter name="parallel" value="true"/>
10    <parameter name="runs" value="10"/>
11    <parameter name="max_time" value="100.0"/>
12    <parameter name="log_interval" value="1.0"/>
13    <parameter name="model_file" value="modelA1.sbml"/>
14    <parameter name="simulation_algorithm" value="tl"/>
15  </parameterSet>
16 </parameters>
~
~
-- INSERT --                               16,14      All

```

FIGURE 8.2: Terminal window running the VIM editor with an open *ngss* XML simulation parameters file.

output (STRING) Sets the type of output for simulation logging:

“csv”, “hdf5”, “console”, “performance”, “gnuplot”.

runs (INTEGER) Number of simulation runs to execute.

max_time (DOUBLE) Amount of *simulation time* to execute model.

max_runtime (DOUBLE) A wall time¹ limit in seconds for the simulator to run.

seed (UNSIGNED LONG INTEGER) Seed to initialise random generator. A seed value of zero will generate a random seed based on system sources of entropy (or system time).

parallel (BOOLEAN) Enables SSA run parallelisation using *OpenMP*.

mpi (BOOLEAN) Enables distributed SSA runs using *OpenMPI*.

show_progress (BOOLEAN) Enables progress logging to the *standard out*. This outputs a (*simulation time*, *run*) tuple after each second of simulation time has been executed by any active run.

compress (BOOLEAN) If *hdf5* output is selected, this option enables *gzip* compression.

¹Wall (clock) time in this context is the human perceived “real world” amount of time that the simulation has been running on the machine.

data_file (*STRING*) If *hdf5* output is selected, this option defines a filename to write to. Otherwise, *hdf5* output is written to the model filename appended with a *hdf5* file extension.

8.4 Software engineering

Ngss was developed in adherence to an object oriented programming style in the multi-paradigm C++ programming language. The primary development environment was Ubuntu Linux 12.04, using the command line *GNU* development toolchain [129] and the *VIM* text editor. The target compiler for Linux was *GCC* 4.6.3 which supports a large subset of the C++11 standard [130] (using the `-std=c++0x` compiler flag). The target compiler for Windows was the Visual Studio 2010 version of the Microsoft C++ compiler (*MSVC10*) which had a smaller subset of C++11 than *GCC* 4.6.3. Therefore, to generate cross-platform source code, I was limited to the intersection of the 2 subsets of C++11 support. The Mac OS X platform supports the *GNU* toolchain (including *g++* 4.6.3) via the *MacPorts* package management system [131].

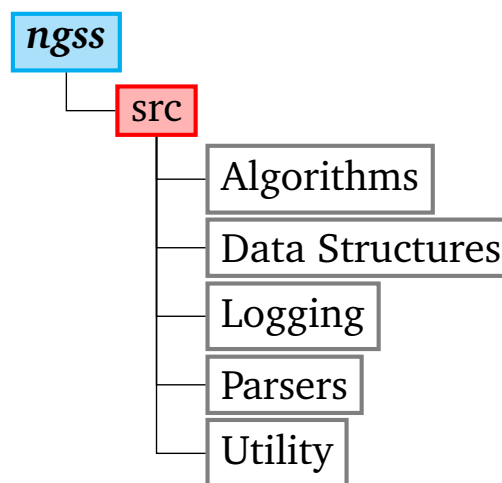


FIGURE 8.3: Diagram of *ngss* file system source tree.

Figure 8.3 shows the directory structure of the *ngss* source code. The *Parsers* folder contains classes to parse simulation parameter files (*CRapidParamsParser*) and SBML model files (*CSBMLReader*). The *Data Structures* folder contains classes that hold data generated by the *parsers*: *CParamsData* and *CModelData*. This folder also contains data structures used by SSA implementations as algorithmic optimisations (e.g. *CDependencyGraph*). The *Algorithms* folder contains the SSA implementation source code, including the *CStochasticSimulationAlgorithm* base class that all SSAs inherit from and the *ISimulationAlgorithm* interface that they must implement. This folder also holds the *CAlgorithmManager* class which provides *SimulateAlgorithm* functions to execute simulation when provided with initialised *CModelData* and *CParamsData* objects. The *Logging* folder provides classes that manage and generate simulator output (e.g. *CCSVOutput* and *CHDF5Output*). These logging classes inherit from the *CDataLogger* base class and must implement *pure virtual functions* declared in the *IDataLogger* interface. The *Utility* folder provides “convenience/wrapper” classes for pseudo-random number generation (*CRandomNumberGenerator*) and high precision wall clock timing for performance evaluation purposes (*CHighResolutionTimer*).

The main function of *ngss* uses the *Read* function of the *CRapidParamsParser* class to generate a *CParamsData* object (by parsing a XML simulation parameters file provided as a command line argument). The biochemical model file name and model parser type are then available from the *CParamsData* object. A model parser object is then initialised (e.g. *CSBMLReader*) and the *GenerateModelData* function is executed (using the model file name), which returns a populated *CModelData* object. If the *CModelData* object is valid, the *CAlgorithmManager* class is used to simulate the model. The *CAlgorithmManager* class exposes *SimulateAlgorithm* (single thread execution), *SimulateAlgorithmOpenMP* (parallel multi-core execution) and *SimulateAlgorithmMPI* (for parallel computing clusters) functions.

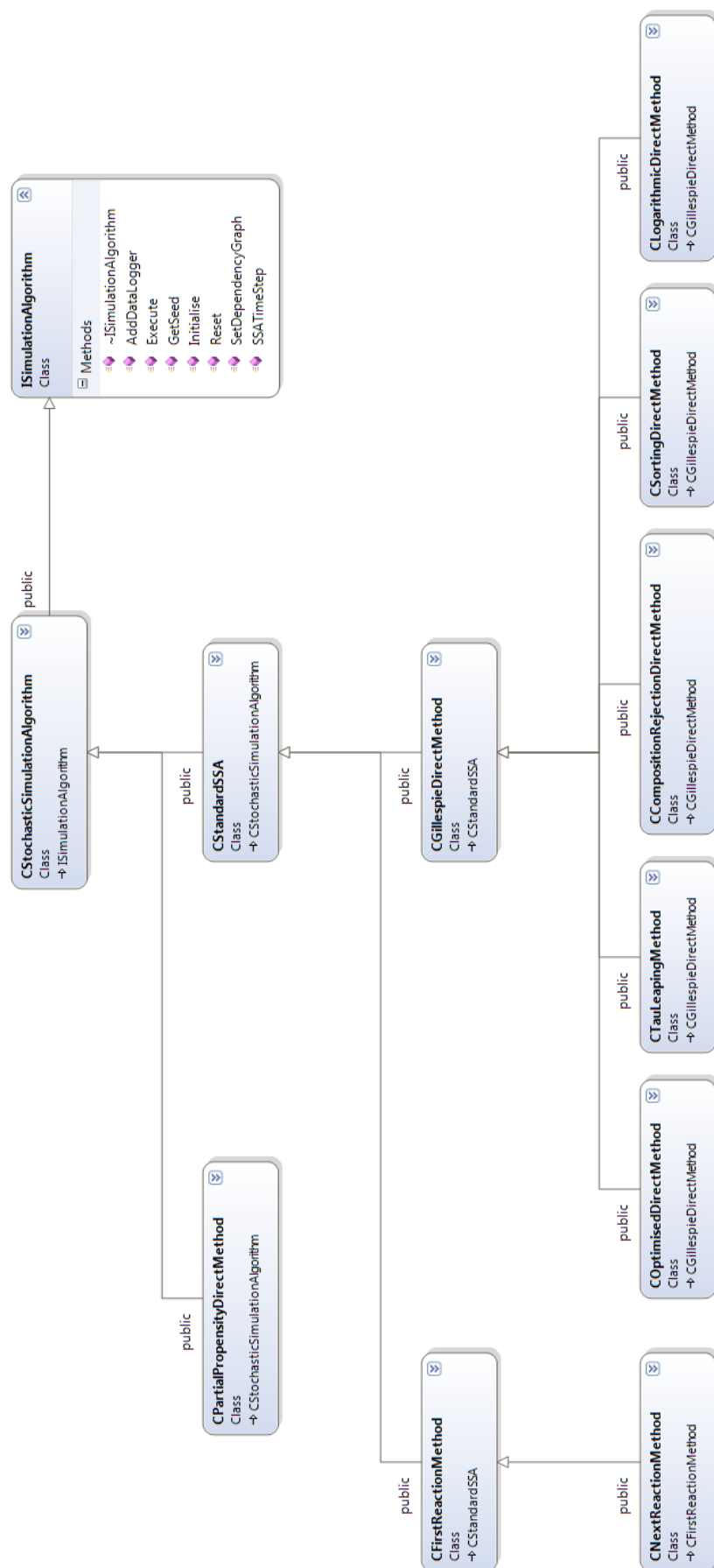


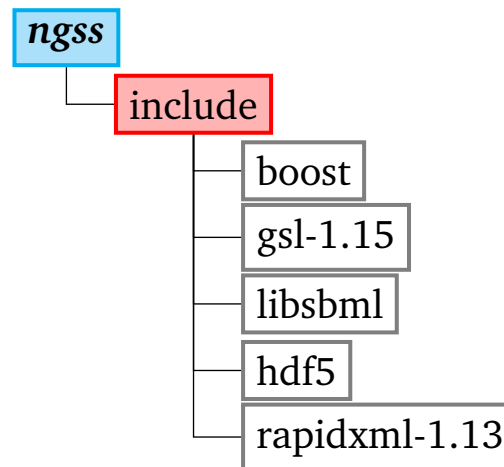
FIGURE 8.4: C++ class diagram of ngss SSA implementations. Arrows indicate an *inheritance* relationship (*subclass* \rightarrow *superclass*). `ISimulationAlgorithm` is an *interface* class that declares C++ pure virtual functions (*Methods*) that all subclasses must implement.

The `SimulateAlgorithm` function calls the `SetAlgorithm` function which returns the SSA selected in the parameters (as an instantiated `ISimulationAlgorithm` object). This dynamic selection of algorithm at runtime is an instance of the *strategy* programming design pattern. Figure 8.4 shows the *inheritance-tree* class diagram for all the `ISimulationAlgorithm` derived classes. ODM, TL, CR, SDM and LDM inherit directly from DM (`CGillespieDirectMethod`). The NRM implementation inherits directly from the FRM implementation. Both DM and FRM inherit from the `CStandardSSA` class which provides “standard” propensity calculation functionality. PDM inherits directly from the `CStochasticSimulationAlgorithm` base class (i.e. not `CStandardSSA`) because it does not use standard propensity functions.

The `IDataLogger` objects are also dynamically instantiated for output functionality (e.g. `CCSVOutput`) and passed to the `ISimulationAlgorithm` object using the `AddDataLogger` function (see Figure 8.4). The `ISimulationAlgorithm::Execute` function performs a complete (single) stochastic run, therefore it is called N times (where N is the number of runs to execute). The `CStochasticSimulationAlgorithm::Execute` function definition calls the *pure virtual* `SSATimeStep` function which must be implemented by every SSA formulation. The `SSATimeStep` is called at every algorithmic iteration to compute reaction execution and populates a `SAAlgorithmEvent struct` with events generated by SSA algorithmic execution on a per iteration basis. The `SAAlgorithmEvent` data is then used to update the species state vector and simulation time.

8.4.1 External software libraries

Figure 8.5 shows the directory tree of the *ngss include* folder. *Ngss* specific header files are contained in the *ngss src* folder (see Figure 8.3), therefore this *include* folder contains header files for external libraries used by *ngss*. In this section, I describe the third party source code & libraries used by the *ngss* simulator (shown in Figure 8.5).

FIGURE 8.5: Diagram of *ngss* *include* directories on the file system.

8.4.1.1 Boost

The *boost* libraries [132] extend the C++ standard library (*STL*) [133] with further features. Use of these libraries is standard for C++ developers, and commonly used *boost* features are integrated into future version of the C++ standard. Ngss development was based on *boost* version 1.46, but the software is compatible with more recent versions of the libraries. The *boost* libraries are released under the terms of the *Boost licence*. This permissive free² open source licence allows for modification, reuse and redistribution so long as the original copyright notices are maintained.

The C++ standard library does not currently include cross-platform support for file system access. Ngss uses ***boost::filesystem*** to: (1) check if model and parameter files exist, (2) delete files (post-run removal of intermediates), (3) create new directories (store intermediates), (4) rename files (manipulating intermediates). The ***boost::tokenizer*** library is used to split command line parameters into separate *tokens* to ease parameter parsing.

8.4.1.2 GSL

The *GNU Scientific Library (GSL)* is a free open source numerical library [134]. *GSL* offers a plethora of functionality for scientific applications including linear algebra,

²Free in this context means “free as in freedom”, but *not* “free as in beer”.

algorithmic methods and statistics. *Ngss* uses version 1.15 of the *GSL*, but should be compatible with more recent versions of the library. *GSL* is released under the terms of the *GNU Public Licence (GPL)* version 3 [125]. Although the *GNU GPL* is a free licence (and allows for modification and redistribution of source code), it is “restrictive” in that derivative works must also be open sourced, and any other code must use a compatible licence. Therefore, GNU GPL libraries may be disregarded by proprietary developers. Crucially, the *GSL* licence forces *ngss* to also be released under a compatible licence.

Ngss employs *GSL* to provide random number generation. The `CRandomNumberGenerator` class uses the `gsl_rng_mt19937` *Mersenne Twister* pseudorandom number generator (PRNG). This PRNG has a “tremendously large” [135] period of $2^{19937} - 1$, which makes it suitable for applications that consume a large amount of random numbers. The *Mersenne Twister* can generate hundreds of millions of pseudorandom numbers per second on a CPU [136], which means it is appropriate for high performance simulations. The *ngss* exact SSA implementations consume random numbers from the *GSL* uniform distribution, whilst TL also samples the *GSL* poisson distribution.

8.4.1.3 libSBML

LibSBML [52] is a free open source library for parsing, writing and validating *Systems Biology Markup Language (SBML)* [33] files. *Ngss* uses *libsbml* version 5.6.0 of the library, but the software is compatible with more recent versions. *Libsbml* is released under the terms of the *Lesser GNU Public Licence (LGPL)* version 2.1. This licence is more permissive than the original *GPL*, and allows source code to be reused, modified and redistributed without requiring derivative work to remain open source if dynamically linked to the executable. This licence also requires attribution to the original author of the library in the derivative works.

The CSBMLReader class employs *libSBML* to parse SBML models using the `readSBML` function. A `CModelData` object is subsequently populated with model information from the SBML file. This object translates the model information into structures that can be executed by the SSA implementations.

8.4.1.4 HDF5

HDF5 (Hierarchical Data Format) [34] is a flexible file format designed for “high volume and complex data” [137]. *Ngss* uses version 1.8.6 of the HDF5 C++ API, but should be compatible with more recent versions. The HDF5 development libraries are released under a *BSD* style licence [138]. This is a permissive licence that only requires attribution to the original developer and redistribution of copyright notices.

HDF5 is a “filesystem-like” data format that defines two major containers: *Datasets* and *Groups*. *Datasets* are laid out as multidimensional arrays and can be grown or shrunk dynamically if required. *Groups* resemble directories in a filesystem [137], and each HDF5 file has a *root Group*. *Groups* can “hold” *Datasets* or other *Groups*. HDF5 *Attributes* can be used to annotate *Groups* or *Datasets*. *Ngss* possesses a `CHDF5Output` class to optionally store SSA time-series data in *HDF5* format. The *root Group* of the *ngss* HDF5 format is annotated with the following *Attribute* meta-data: (1) `simulator name`, (2) `simulator version`, (3) `model file`, (4) `runs` (number of runs), (5) `log interval`, (6) `max time` (simulation time), (7) `simulation algorithm` and (8) `seed` (random seed). For each performed simulation run, a “run *Group*” is added to the *root Group*. A two dimensional *Dataset* of unlimited size is created within each *run Group* which holds species amounts time-series data (see `LogEvent` function). The `writeRunSpeciesNames` function also records the names and indices of the species written to the time-series data in another *Dataset*.

I apply two *filters* to the HDF5 file if the simulation parameters compression setting is set to *true* (default value is *true*): (1) ***Deflate*** & (2) ***Shuffle***. The *Deflate* filter applies *gzip* on-the-fly compression [139]. I use the lowest level of *Deflate* compression as I have found that increased settings provide diminished returns at greater

computational cost. I have also found that applying compression actually improves computational performance whilst reducing file space requirements. Therefore, the computational overhead of the lowest compression setting is offset by the resultant saving in I/O operations. The *Shuffle* filter rearranges values in the data stream in order to improve compression ratios. Data is written as *chunks*, where a chunk is an “atomic” [139] unit for I/O operations. After a small benchmark of different configurations, I found that setting a chunk size of 10KB provides good I/O performance for *ngss*.

8.4.1.5 RapidXml

RapidXml is a free open source library for XML parsing [128] and can be licensed under the Boost or MIT licences. Both of these licences are permissive, only requiring attribution to the author and copyright notices to be maintained. *RapidXml* is “header only” which means that its entire implementation is distributed in C++ header files. Therefore, the library can be fully integrated into the source code of *ngss* and thus does not require separate compilation. *RapidXml* version 1.13 is distributed as part of the *ngss* source code. This lightweight library claims to be up to 100 times faster than other mainstream XML parsing libraries [140].

The `CRapidParamsParser` class uses *RapidXml* to parse *ngss* simulation parameters (which are stored in XML format) and populates a `CParamsData` object. The `CParamsData` object is available to many different classes in the simulator source code.

8.4.2 Parallelising stochastic runs

Scientists usually require the execution of multiple SSA runs in order to determine average system behaviour or to generate other statistics about the system trajectories. It may also be necessary to perform a very large number of runs in order to detect rare system events. For example, one would need to execute a model 10^{11} times to

estimate the probability an event that has a 10^{-7} likelihood of occurring with a 95% confidence interval [141].

Executing multiple stochastic simulation runs simultaneously is an “embarrassingly parallel” procedure. This means that each run can be executed independently with no intercommunication required during simulation. All runs have the same initial conditions (as dictated by the model), but would each possess a different pseudo-random generator seed and generate a different simulation trajectory. Therefore, one can distribute these independent simulation runs on different CPU cores (see Section 8.4.2.1) or different machines in a computing cluster (see Section 8.4.2.2). *Ngss* uses *OpenMP* and *OpenMPI* to take advantage of SSA’s direct mapping to task parallelism. This is enabled at runtime using the `parallel` and `mpi` simulation parameters. After simulation completes, the simulator output (generated by *IData-Logger* derived classes) may need to be combined.

8.4.2.1 OpenMP

```

void CAlgorithmManager::SimulateAlgorithmOpenMP( const CParamsData &params,
                                                  const CModelData *pModelData )
{
    const int RUNS = params.GetRuns();
    #pragma omp parallel for
    for ( int i = 0; i < RUNS; ++i )
    {
        ISimulationAlgorithm *pAlgorithm = SetAlgorithm( params );
        pAlgorithm->Execute( params );
    }
}

```

FIGURE 8.6: Compressed code fragment from the *ngss* `CAlgorithmManager::SimulateAlgorithmOpenMP` function.

OpenMP is a cross platform API that supports scalable *shared-memory* parallel programming [35]. Employing *OpenMP* means that *ngss* can dynamically (and automatically) distribute stochastic runs to each core of a multi-core CPU that has a single shared pool of global memory for a process. This is significant as modern CPUs are increasingly multi-core in order to improve thermo-efficiency.

The `CAlgorithmManager::SimulateAlgorithmOpenMP` function is the entry point for *ngss OpenMP* execution. Figure 8.6 shows a fragment of this function, but as it removes most of the source code it should only be considered as a pseudo code description of functionality. One can see that simulation algorithm is instantiated and executed using the `for` loop. Enlisting *OpenMP* task parallelism is as simple as adding a `#pragma` compiler directive above the `for` loop that requires parallelisation.

8.4.2.2 OpenMPI

```
void CAlgorithmManager::SimulateAlgorithmMPI( const CParamsData &params,
                                              const CModelData *pModelData )
{
    MPI_Comm_rank( MPI_COMM_WORLD, &nProcess );
    MPI_Comm_size( MPI_COMM_WORLD, &nProcessTotal );

    ISimulationAlgorithm *pAlgorithm = SetAlgorithm( params );
    pAlgorithm->Execute( params );

    if ( nProcess == MASTER )
    {
        int nCount = nProcessTotal - 1;
        while ( nCount > 0 )
        {
            int nProcDone = -1;
            MPI_Status status;
            MPI_Recv( &nProcDone, 1, MPI_INT, MPI_ANY_SOURCE,
                     MSG_COMPLETE, MPI_COMM_WORLD, &status );
            nCount--;
        }
        //now call functions to collate and process data
    }
    else
    {
        //let master know we have finished
        MPI_Send( &nProcess, 1, MPI_INT, MASTER, MSG_COMPLETE, MPI_COMM_WORLD );
    }
}
```

FIGURE 8.7: Compressed code fragment from the *ngss* `CAlgorithmManager::SimulateAlgorithmMPI` function.

OpenMPI is an open source implementation of the Message Passing Interface (MPI) [36]. *OpenMPI* is an API that provides access to a protocol for communication on *distributed-memory* architectures. *Ngss* employs *OpenMPI* to enable distribution of SSA runs on high performance computing clusters.

Ngss uses a “master/slave” model, assigning the status of “master” to the first SSA process (all other processes are “slaves”). The *master* instance waits for all *slave* instances to send task completion messages before collating and processing generated stochastic run data. Figure 8.7 shows a fragment of this function (with most source code removed) to elucidate this feature.

Chapter 9

Conclusions

9.1 Summary of thesis motivation

Scientists in the fields of systems and synthetic biology use computational techniques to measure, decipher and comprehend complex biological systems. Simulation is an important tool for computational hypothesis testing that is traditionally performed by evaluating deterministic ordinary differential equation models. However, this does not account for the *stochastic* noise present in cellular biosystems or accurately reproduce the *discrete* switching behaviour found in gene regulatory networks. Stochastic simulation algorithms can generate *exact* system trajectories but may become computationally intractable with large or detailed models. Thus, poor computational performance of stochastic simulation algorithms may impede the knowledge discovery afforded by this era of high-throughput cell biology in spite of increasing computational power. The field of synthetic biology aims to design large biosystems from defined genetic components. Therefore, synthetic biologists require innovation in SSA technology for hypothesis testing in order to avoid costly wet lab trial and error.

After evaluating a range of algorithmic advancements in the SSA, I found that so called “state-of-the-art” SSA formulations could be outperformed by more primitive methods for certain classes of model. Furthermore, I found that sets of models with

similar characteristics would perform favourably with a particular subset of SSA formulations, whilst other model types would favour a distinct subset of algorithms. From a review of the literature, it is difficult for a scientist to identify the fastest SSA for their particular model. I also found that the computational performance difference may vary by several orders of magnitude between fast and slow SSA formulations. Therefore, it is important that a scientist is able to automatically deduce the fastest SSA for a given model *a priori* to simulation.

9.2 Evaluation of hypotheses

This thesis set out to evaluate 3 hypotheses related to SSA performance.

Hypothesis 1

There is no single SSA that is superior in performance for every biomodel

To test the first hypothesis of this thesis, one would have to find evidence that a particular SSA was fastest over all models. Chapters 3 & 5 benchmarked a total of 388 different biochemical models. Whilst I found that certain modern SSAs had better overall performance than the original (FRM & DM) formulations, there was no SSA found that was fastest for all models. Statistical tests to rank algorithm performance (see Table 5.4) show that the 3 highest ranked algorithms had very similar mean rankings. Furthermore, it is not sufficient to simply select an algorithm that has a high mean ranking, because my analysis did show that all SSAs have subsets of models that another formulation is better suited to (see Figure 5.7 and Table 5.2).

Hypothesis 2

There is a relationship between biomodel characteristics and SSA performance

The second hypothesis is closely related to the third hypothesis, as a prediction of the fastest SSA for an arbitrary model relies on a relationship between performance and

model characteristics. Chapter 4 investigated the network analysis of biochemical models as a metric of model characteristics, surveying a large number of graph properties. The combination of model metric and performance benchmarking data allowed me to test both the second and third hypotheses.

Hypothesis 3

An algorithm can select the best SSA for an arbitrary model with only a small margin of error

Chapter 6 presented the use of statistical classification techniques to predict the fastest SSA for an arbitrary model (using graph property values derived from the model). I found that I was able to predict the fastest SSA with up to 65% accuracy based on model topological properties compared to a probability of $\frac{1}{9}$ for a blind random selection of SSA (see Table 6.5). Furthermore, I found that selection accuracy of up to 63% could still be achieved with a subset of *fast-to-compute* properties, indicating feature redundancy in the global set of properties. Introducing a relaxation threshold (ϵ) that allows any algorithm prediction within 10% of the fastest algorithm to be considered successful resulted in a prediction accuracy of 85% for *fast-to-compute* properties with the Linear SVC predictor.

Therefore, I have demonstrated that an algorithm can indeed select the fastest SSA for an arbitrary model with good accuracy. Furthermore, this prediction is performed using model characteristics, implying that there must be a relationship between model characteristics and algorithm performance.

9.3 Knowledge transfer

The prediction analysis performed in this thesis was directly applicable for real world application to improve SSA usability. Chapter 7 detailed the “*ssapredict*” tool created based on the finding of this thesis. This easy-to-use web application improves accessibility of the SSA for biologists, providing an accurate prediction of SSA performance and providing simulation capabilities via the *ngss* simulator.

Chapter 8 described the “*ngss*” simulator which is based on the SSA implementations tested and benchmarked in the SSA benchmarking suite (see Chapter 4). *Ngss* is available as part of the *ssapredict* suite, and has also been integrated into the *Infobiotics Workbench 2* software that is currently under development as part of the *EPSRC ROADBLOCK project grant (EP/I031642/1)*.

9.4 Limitations & reflections

The meta-simulation technique presented in this thesis makes predictions of the fastest SSA based on the *static* topological properties of a model. However, stochastic simulations, by their very nature, are subject to dynamic changes in system behaviour. Figure 3.14 shows that SSA performance profiles can vary greatly depending on the transient system state. As the analysis presented in this thesis is based on the model topological properties of *unweighted* graphs derived from partially parametrised biosystems (see Section 4.2.2), it is not possible to capture or account for this behaviour.

There is a distinct lack of fully parametrised stochastic models available or catalogued in the appropriate literature due to the established prevalence of deterministic ODE models. In order to generate performance analysis that can capture a full account of discrete stochastic system behaviour, one requires the curation of a large number of fully parametrised stochastic models. The low adoption rate of stochastic modelling (compared to deterministic ODE modelling) is problematic in and of itself. As shown in model A7 (see Section 3.2.8), discrete systems with large actor/agent populations that scientists often assume can be evaluated using continuous deterministic approaches may be profoundly affected by discrete stochastic system fluctuations. Furthermore, model A1 (see Section 3.2.2) showed that biological systems can in fact rely on stochastic noise to, *almost counter-intuitively*, implement robustness in biosystem behaviour. This raises important questions about the validity of some biochemical analyses based on continuous deterministic approaches. Is it possible that

scientists are *underfitting* models of biological systems by avoiding the use of discrete stochastic biochemical modelling & simulation paradigms?

9.5 Future research directions

9.5.1 Online meta-SSA

The logical next step in this course of research is to develop a meta-simulation technique that can dynamically adjust SSA formulation at runtime based on transient system state. With access to a large dataset of fully parametrised stochastic models, one would be able to determine the impact of graph weighting on prediction quality. One could dynamically weight the model-derived RDG with with current propensities values and the SDG with computed partial-propensity values. Predictor training data would be generated by performing multiple SSA runs, taking “*snapshots*” of algorithm performance for a range of graph propensity weightings. This training methodology should incorporate a method to produce a set of samples for each model that maximises the variation in SSA performances and attempt to find “winning” graph propensity weighting configurations for every SSA if possible.

9.5.2 Increasing SSA adoption

As previously discussed, the SSA is typically underutilised for biochemical modelling. The *ssapredict* software produced as a contribution of this thesis aims increase SSA accessibility by focussing on *ease-of-use* and providing simulation functionality for an uploaded model. Future research needs to investigate exactly why scientists overlook the SSA, including an anthropological study of biochemical modelling. In place of speculation, a comprehensive appraisal of SSA usability should generate research targets that improve the adoption of stochastic modelling and simulation.

9.5.3 Scaling up the SSA

Biological models are becoming more finely detailed and intricate to match reality, but this comes at the detriment of computational performance when executing such systems with the SSA. Furthermore, scientists increasingly wish to simulate models of large systems such as biofilm formation involving millions of bacterial cells. Whilst this thesis has introduced techniques to improve computational performance by ascertaining which algorithm is most applicable to a particular class of model, this is not a solution that would allow the SSA to cope with ultra-large systems that scientists may wish to investigate.

The SSA is inherently difficult to parallelise for a single algorithmic instance but has natural parallel independence of individual runs [23]. However, a model of especially high complexity may be intractable for a single run. Biological systems are composed of cells, which encapsulate stochasticity at the micro-scale yet are elements of large systems at the macro-scale. Therefore, it is possible to parallelise SSA runs within a single large model composed of many cells by treating each cell as an independent stochastic simulation. This would be an agent-based system where a global timestep can be passed to each simulation run to synchronise the system. Furthermore, interaction between cells can be handled by a physics engine to generate a physically realistic model that could include soft-body simulation and fluid dynamics. Other forms of cellular inter-process communication such as quorum sensing and conjugation would also be available. Several groups have embarked upon using deterministic agent based models with rigid-body physics to simulate bacterial colonies [142–144]. These simulations integrate some elements of stochasticity but an important “next step” is to integrate the SSA to generate more realistic simulations of large scale biological systems.

Appendix A

Statistical Methods

A.1 Pearson product moment correlation coefficient

The Pearson product moment correlation coefficient r is a measure of the linear relationship between two variables X and Y [145]. The formula for calculating the coefficient r is shown below:

$$r = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^n (X_i - \bar{X})^2} \sqrt{\sum_{i=1}^n (Y_i - \bar{Y})^2}} \quad (\text{A.1})$$

The calculated Pearson correlation coefficient r will be a value in the range $[-1, +1]$. A value of $r = +1$ indicates a perfect positive linear relationship between the variables X and Y , whilst $r = -1$ reveals a perfect inverse relationship. Values of r close to 0 imply that there is no linear relationship between the two variables.

A.2 Mann-Whitney U test

The Mann-Whitney U test [146] is an ordinal non-parametric measure of the similarity of two random variables x and y . The null hypothesis of the test is that the two samples are equivalent. Non-parametric statistical measures do not require a fixed parameter set or specific probability distribution. The test relies on the calculation of a U statistic introduced by Wilcoxon [147]:

$$U = mn + \frac{m(m+1)}{2} - T \quad (\text{A.2})$$

The T statistic is calculated as the sum of the *ranks* of y given the sorted sequence of x and y . The terms m and n are the sample sizes of x and y respectively. The U statistic can be calculated for both samples by switching which are represented by x and y . The significance of the result can be calculated from the sample size and U statistic.

A.3 Kruskal-Wallis H test

The Kruskal-Wallis H test [148] is an ordinal non-parametric measure of the similarity of group mean rankings. This test can be considered an extension to the Mann-Whitney U test (see Appendix A.2) to compare more than two groups. The null hypothesis of the test is that the mean *rankings* of the groups come from the same distribution. The test relies on the calculation of a H statistic [149]:

$$H = \frac{12}{N(N+1)} \left(\sum \frac{(\mathbf{T}_g)^2}{n_g} \right) - 3(N+1) \quad (\text{A.3})$$

The first step is to rank the combined data from all the groups. N is the total number of samples of the combined group data. \mathbf{T}_g is the sum of the ranks of a data from a

group g . n_g is the number of samples from a group g . A p-value can be computed by sampling the chi-squared distribution.

A.4 Shapiro-Wilk test

The Shapiro-Wilk test [150] is a test of population normality. The null hypothesis of this test is that the population is normally distributed. The test relies on the calculation of a W statistic [151]:

$$W = \frac{[\sum_{i=1}^{\lfloor n/2 \rfloor} a_{n-i+1}(y_{n-i+1} - y_i)]^2}{\sum_{i=1}^n (y_i - \bar{y})^2} \quad (\text{A.4})$$

The a_{n-i+1} variable is calculated using tabulated coefficients from the original 1965 paper [150]. If the calculated p-value is less than the chosen alpha value, the null hypothesis is rejected and the population is *not* normally distributed.

However, the original Shapiro-Wilk test is only suitable for populations with up to 50 samples. Royston extended the test to deal with large sample sizes (up to $n = 2000$) [152]. This extended test was computationally expensive because of the requirement of large matrix manipulations [153]. Furthermore, there were no guarantees given regarding the accuracy of the updated formulation. Approximately a decade later, Royston introduced an approximated version of the Shapiro-Wilk test that has widespread adoption in modern statistical software packages such as R [153, 154]:

$$W = \frac{(\sum a_i y_i)^2}{\sum (y_i - \bar{y})^2} \quad (\text{A.5})$$

The a variable is approximated/estimated such that $a = (a_1, \dots, a_n)$ where $(n - 1)^{-\frac{1}{2}} \sum a_i y_i$.

A.5 Spearman's rho rank correlation test

The Spearman's rank correlation test (also known as Spearman's rho) is a non-parametric measure of the relationship between two *ranked* variables x and y . It can be used as a replacement for the Pearson product moment correlation coefficient (see Appendix A.1) when a non-parametric statistic is required. The test measures how *monotonic* the relationship is between the two variables. A monotonic relationship is when the variables *only* either have increasing *or* negative relationship (and the relationship direction does not change). A value of $\rho = +1$ indicates a perfect positive monotonic relationship between the variables X and Y , whilst $\rho = -1$ reveals a perfect negative monotonic relationship.

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \quad (\text{A.6})$$

The variable d is the difference between the ranking variables ($d_i = x_i - y_i$).

Appendix B

Statistical Classification

B.1 Linear regression

Linear regression is a method to model a linear relationship between a “*response*” variable and one or more “*controlled*” variables. For a single controlled variable x , one generates a linear equation $\hat{y} = a + bx$ which finds the value of the response variable \hat{y} with the smallest error possible [155]. Least squares estimation is used to minimise errors for fitting a linear data relationship. To be precise, the sum of square of the errors (between fitted value \hat{y}_i and respective data point y_i) is minimised for all data points:

$$\text{Sum of squares} = \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (\text{B.1})$$

The coefficient of determination r^2 can be calculated for linear regression models. This gives an indication of the quality of the fit between the model and data points:

$$r^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2} \quad (\text{B.2})$$

Linear regression is a type of *predictor*, it will generate a response variable for a set of controlled variables. However, one can use linear regression as a classifier by using the fitted response variables for rankings.

B.2 Linear Support Vector Classifier

Linear support vector classifier (LinearSVC) is an implementation of *support vector machines (SVM)* using a *linear kernel*. SVM [156] is a supervised learning technique. Supervised learning requires data that has been *labelled* as samples of the target classes.

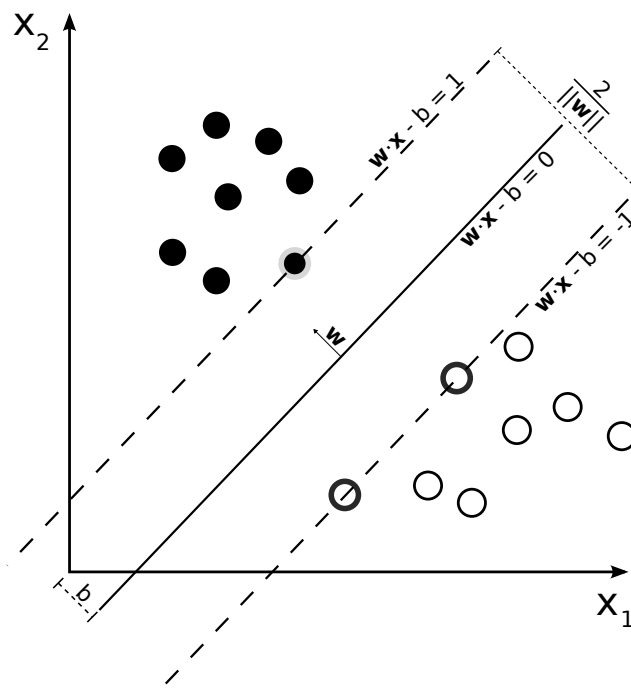


FIGURE B.1: Optimal hyperplane (solid line) and margins (dashed lines) for SVM trained with 2 classes (class samples shown as circles in plot). Taken from [157].

SVM computes an *optimal hyperplane(s)* through a dataset, which separates the classes. A hyperplane is a $(n - 1)$ -dimensional subspace of the n -dimensions of the dataset. An *optimal hyperplane* is a hyperplane that divides the space with *maximal* margin (separation) between the classes. Figure B.1 shows an optimal hyperplane (shown as a solid line) through a (hypothetical) SVM trained dataset containing

data samples from 2 classes (shown as circles). Two parallel vectors (dashed lines) demarcate the maximal margins between the hyperplane and the classes. The subset of class samples that lie on the margins are known as the *support vectors*.

A hyperplane that separates classes is represented by the equation [158]:

$$f(x) \equiv \mathbf{w} \cdot \mathbf{x} - b = 0 \quad (\text{B.3})$$

Thus to use this equation as the *decision rule*, one needs to find the normal vector \mathbf{w} and offset b . These values can be found using quadratic programming.

B.3 Logistic Regression

Logistic regression [159] is a binary classification method meaning it can predict 2 classes $y \in \{0, 1\}$ given a set of controlled (i.e. feature) variables. This method is closely related to linear regression (see Appendix B.1) but instead uses a sigmoid function for the linear model:

$$\ln \left(\frac{p(x)}{1 - p(x)} \right) = a + bx \quad (\text{B.4})$$

Instead of measuring a y variable (as in linear regression), the probability p of a particular class is computed. The linear model is fitted using the maximum-likelihood method. Logistic regression can be extended to become *multi-class* so that it can predict 3 or more discrete class values. This can be achieved using a *one-versus-rest* scheme where the classifier is run k times (where k is the number of classes). This means that standard logistic regression is performed for each class in the feature space whilst treating the all other classes as single class.

B.4 k-Nearest Neighbour Classification

k-Nearest Neighbour (k-NN) [160] classifier is a simple non-parametric instance-based “*lazy learning*” method [161]. The principle behind this method is to compare an input sample \mathbf{q} to training data \mathcal{D} and return the k most similar instances. The class of \mathbf{q} is then determined from the class ownership of the returned instances. The closest neighbours can be judged by iterating over the training examples and measuring feature *distance*, for example using the Minkowski distance:

$$d(\mathbf{q}, x) = \left(\sum_{i=1}^n |\mathbf{q}_i - x_i|^p \right)^{1/p} \quad (\text{B.5})$$

where $x_i \in \mathcal{D}$ and $p = 2$ is equivalent to Euclidean distance. The predicted class of the input sample can be computed by simply returning a (uniform) majority vote, or for finer accuracy using a weighted distance voting scheme:

$$\text{vote}(y_j) = \sum_{c=1}^k \frac{1}{d(\mathbf{q}, x_c)^2} 1(y_j, y_c) \quad (\text{B.6})$$

where class y_j is assigned a vote by the neighbour x_c . $1(y_j, y_c)$ returns 1 if the class labels match and 0 if not. The value of n affects the influence of distant neighbours [162].

B.5 k-fold Cross-validation

k -fold Cross-validation is method to evaluate the quality (i.e. the accuracy) of a classifier for a dataset \mathcal{D} [163]. The principle behind k -fold cross-validation is to assign training and test sets to a dataset and ensure that each sample is evaluated once. The dataset \mathcal{D} is first partitioned into k mutually exclusive subsets \mathcal{D}_t where $t = \{1, 2, \dots, k\}$. The classifier is tested on a *fold* \mathcal{D}_t using $\mathcal{D} \setminus \mathcal{D}_t$ as the training set. Each of the k folds are evaluated in turn to ensure all samples have been evaluated.

The mean accuracy (and standard deviation) of the k classifier evaluations can then be generated.

Bibliography

- [1] Alan M Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society*, 327:37–72, 1952.
- [2] Hiroaki Kitano. Systems biology: A brief overview. *Science*, 295(5560):1662–1664, 2002.
- [3] SA Benner and AM Sismour. Synthetic biology. *Nature reviews. Genetics*, 6(7): 533–543, 2005.
- [4] Christian L Barrett, Tae Yong Kim, Hyun Uk Kim, Bernhard Ø Palsson, and Sang Yup Lee. Systems biology as a foundation for genome-scale synthetic biology. *Current opinion in biotechnology*, 17(5):488–492, 2006.
- [5] Ernesto Andrianantoandro, Subhayu Basu, David K Karig, and Ron Weiss. Synthetic biology: new engineering rules for an emerging discipline. *Molecular systems biology*, 2(1), 2006.
- [6] Christina D Smolke. Building outside of the box: iGEM and the BioBricks Foundation. *Nature biotechnology*, 27(12):1099–1102, 2009.
- [7] Daniel T. Gillespie. Stochastic Simulation of Chemical Kinetics. *Annual Review of Physical Chemistry*, 58(1):35–55, 2007.
- [8] Tamas Szekely and Kevin Burrage. Stochastic Simulation in Systems Biology. *Computational and Structural Biotechnology Journal*, 12(20):14–25, 2014.
- [9] Robert C Hilborn, Benjamin Brookshire, Jenna Mattingly, Anusha Purushotham, and Anuraag Sharma. The Transition between Stochastic and Deterministic Behavior in an Excitable Gene Circuit. *PloS ONE*, 7(4), 2012.
- [10] Daniel T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403 – 434, 1976.
- [11] Daniel T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *The Journal of Chemical Physics*, 115(4):1716–1733, 2001.
- [12] Jamie Twycross, Leah Band, Malcolm Bennett, John King, and Natalio Krasnogor. Stochastic and deterministic multiscale models for systems biology: an auxin-transport case study. *BMC Systems Biology*, 4(1):34, 2010.

- [13] Florine Dupeux, Julia Santiago, Katja Betz, Jamie Twycross, Sang-Youl Park, Lesia Rodriguez, Miguel Gonzalez-Guzman, Malene Jensen, Natalio Krasnogor, Martin Blackledge, Michael Holdsworth, Sean R Cutler, Pedro L Rodriguez, and José Antonio Márquez. A thermodynamic switch modulates abscisic acid receptor sensitivity. *The EMBO journal*, 30:4171–4184, 2011.
- [14] A. Moya, N. Krasnogor, J. Pereto, and A. Latorre. Goethe’s Dream. Challenges and Opportunities for Synthetic Biology. *EMBO reports*, 10, S1,:28–32, 2009.
- [15] M. Porcar, A. Danchin, V. de Lorenzo, V.A. dos Santos, N. Krasnogor, S. Rasmussen, and A. Moya. The Ten Grand Challenges of Synthetic Life. *Systems and Synthetic Biology*, 5(1-2):1–9, 2011.
- [16] James Smaldon, Francisco Romero-Campero, Francisco Fernández Trillo, Marian Gheorghe, Cameron Alexander, and Natalio Krasnogor. A computational study of liposome logic: towards cellular computing from the bottom up. *Systems and Synthetic Biology*, 4:157–179, 2010.
- [17] Leong T Lui, Xuan Xue, Cheng Sui, Alan Brown, David I Pritchard, Nigel Halliday, Klaus Winzer, Steven M Howdle, Francisco Fernandez-Trillo, Natalio Krasnogor, et al. Bacteria clustering by polymers induces the expression of quorum-sensing-controlled phenotypes. *Nature chemistry*, 5(12):1058–1065, 2013.
- [18] Hongqing Cao, Francisco Romero-Campero, Stephan Heeb, Miguel Cámara, and Natalio Krasnogor. Evolving Cell Models for Systems and Synthetic Biology. *Systems and Synthetic Biology*, 4:55–84, 2010.
- [19] Goksel Misirli, Jennifer Hallinan, and Anil Wipat. Composable Modular Models for Synthetic Biology. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 11(3):22, 2014.
- [20] JS Hallinan, O Gilfellow, G Misirli, and A Wipat. Tuning Receiver Characteristics in Bacterial Quorum Communication: An Evolutionary Approach using Standard Virtual Biological Parts. In *Computational Intelligence in Bioinformatics and Computational Biology, 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.
- [21] Jonathan Blakes, Jamie Twycross, Francisco Jose Romero-Campero, and Natalio Krasnogor. The Infobiotics Workbench: an integrated in silico modelling platform for Systems and Synthetic Biology. *Bioinformatics*, 2011.
- [22] Francisco José Romero-Campero, Jamie Twycross, Miguel Cámara, Malcolm Bennett, Marian Gheorghe, and Natalio Krasnogor. Modular assembly of cell systems biology models using P systems. *International Journal of Foundations of Computer Science*, 20(3):427–442, 2009.
- [23] Daven Sanassy, Jonathan Blakes, Jamie Twycross, and Natalio Krasnogor. Improving Computational Efficiency in Stochastic Simulation Algorithms for Systems and Synthetic Biology. In *SynBioCCC. 11th European Conference on Artificial Life*, 2011.

- [24] Daven Sanassy, Paweł Widera, and Natalio Krasnogor. Meta-Stochastic Simulation of Biochemical Models for Systems and Synthetic Biology. *ACS Synthetic Biology*, 4(1):39–47, 2015.
- [25] Michael A. Gibson and Jehoshua Bruck. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *The Journal of Physical Chemistry A*, 104(9):1876–1889, 2000.
- [26] Yang Cao, Hong Li, and Linda Petzold. Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. *The Journal of Chemical Physics*, 121(9):4059–4067, 2004.
- [27] James M. McCollum, Gregory D. Peterson, Chris D. Cox, Michael L. Simpson, and Nagiza F. Samatova. The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behavior. *Computational Biology and Chemistry*, 30(1):39 – 49, 2006.
- [28] H. Li and L. Petzold. Logarithmic direct method for discrete stochastic simulation of chemically reacting systems. Technical report, Department of Computer Science, University of California: Santa Barbara, 2006.
- [29] Rajesh Ramaswamy, Nelido Gonzalez-Segredo, and Ivo F. Sbalzarini. A new class of highly efficient exact stochastic simulation algorithms for chemical reaction networks. *The Journal of Chemical Physics*, 130(24):244104, 2009.
- [30] Alexander Slepoy, Aidan P. Thompson, and Steven J. Plimpton. A constant-time kinetic Monte Carlo algorithm for simulation of large biochemical reaction networks. *The Journal of Chemical Physics*, 128(20):205101, 2008.
- [31] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.
- [32] M. Guidry. Algebraic stabilization of explicit numerical integration for extremely stiff reaction networks. *Journal of Computational Physics*, 231:5266–5288, June 2012.
- [33] M. Hucka et al. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.
- [34] Mike Folk, Albert Cheng, and Kim Yates. HDF5: A file format and I/O library for high performance computing applications. In *Proceedings of Supercomputing*, volume 99, 1999.
- [35] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.
- [36] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open MPI: A flexible high performance MPI. In *Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2006.

- [37] Richard Dawkins. *The Selfish Gene*. Oxford University Press, 2006.
- [38] John J Tyson and Bela Novak. Control of cell growth, division and death: information processing in living cells. *Interface Focus*, 4(3):20130070, 2014.
- [39] Nataly Kravchenko-Balasha, Alexander Levitzki, Andrew Goldstein, Varda Rotter, A Gross, Françoise Remacle, and RD Levine. On a fundamental structure of gene networks in living cells. *Proceedings of the National Academy of Sciences*, 109(12):4702–4707, 2012.
- [40] Wikimedia Commons ('LadyofHats'). "Ribosome mRNA translation", 2009. Public Domain.
- [41] Martin Hemberg and Mauricio Barahona. Perfect Sampling of the Master Equation for Gene Regulatory Networks. *Biophysical journal*, 93(2):401–410, 2007.
- [42] Jasmin Fisher and Thomas A Henzinger. Executable cell biology. *Nature Biotechnology*, 25(11):1239–1249, November 2007.
- [43] Corrado Priami. Algorithmic systems biology. *Communications of the ACM*, 52:80–88, May 2009.
- [44] Desmond J Higham. An Algorithmic Introduction to Numerical Simulation of Stochastic Differential Equations. *SIAM review*, 43(3):525–546, 2001.
- [45] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011. ISBN 978-0-321-57351-3.
- [46] Yang Cao, Daniel T. Gillespie, and Linda R. Petzold. Efficient step size selection for the tau-leaping simulation method. *The Journal of Chemical Physics*, 124(4):044109, 2006.
- [47] Yang Cao, Daniel T. Gillespie, and Linda R. Petzold. Avoiding negative populations in explicit Poisson tau-leaping. *The Journal of Chemical Physics*, 123(5):054104, 2005.
- [48] Sagar Indurkha and Jacob Beal. Reaction Factoring and Bipartite Update Graphs Accelerate the Gillespie Algorithm for Large-Scale Biochemical Systems. *PloS ONE*, 5(1):e8125, 2010.
- [49] Jacob Beal, Andrew Phillips, Douglas Densmore, and Yizhi Cai. High-level programming languages for biomolecular systems. In *Design and Analysis of Biomolecular Circuits*, pages 225–252. Springer New York, 2011.
- [50] Alvin Tamsir, Jeffrey J. Tabor, and Christopher A. Voigt. Robust multicellular computing using genetically encoded NOR gates and chemical 'wires'. *Nature*, 469(7329):212–215, 2011.

- [51] Sergi Regot, Javier Macia, Nuria Conde, Kentaro Furukawa, Jimmy Kjellen, Tom Peeters, Stefan Hohmann, Eulalia de Nadal, Francesc Posas, and Ricard Sole. Distributed biological computation with multicellular engineered networks. *Nature*, 469(7329):207–211, 2011.
- [52] Benjamin J. Bornstein, Sarah M. Keating, Akiya Jouraku, and Michael Hucka. LibSBML: an API Library for SBML. *Bioinformatics*, 24(6):880–881, 2008.
- [53] Sydney Brenner. Theoretical biology in the third millennium. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 354(1392):1963–1965, 1999.
- [54] Jongrae Kim, Pat Heslop-Harrison, Ian Postlethwaite, and Declan G Bates. Stochastic Noise and Synchronisation during Dictyostelium Aggregation Make cAMP Oscillations Robust. *PLoS Computational Biology*, 3(11):2190–2198, 2007.
- [55] Hana El-Samad, Mustafa Khammash, Hiroyuki Kurata, and John C Doyle. Feedback regulation of the heat shock response in E. coli. In *Proceedings of the 40th IEEE Conference on Decision and Control, 2001.*, volume 1, pages 837–842 vol.1, 2001.
- [56] Jun Li, Liang Wang, Yoshifumi Hashimoto, Chen-Yu Tsao, Thomas K Wood, James J Valdes, Evangelhos Zafiriou, and William E Bentley. A stochastic model of Escherichia coli AI-2 quorum signal circuit reveals alternative synthesis pathways. *Molecular systems biology*, 2(1):67, 2006.
- [57] Domitille Heitzler, Guillaume Durand, Nathalie Gallay, Aurelien Rizk, Seungkirl Ahn, Jihee Kim, Jonathan D Violin, Laurence Dupuy, Christophe Gauthier, Vincent Piketty, Pascale Crepieux, Anne Poupon, Frederique Clement, Francois Fages, Robert J Lefkowitz, and Eric Reiter. Competing G protein-coupled receptor kinases balance G protein and β -arrestin signaling. *Molecular Systems Biology*, 8(1):590, 2012.
- [58] Nadav M Shnerb, Yoram Louzoun, Eldad Bettelheim, and Sorin Solomon. The importance of being discrete: Life always wins on the surface. *Proceedings of the National Academy of Sciences*, 97(19):10322–10324, 2000.
- [59] Andrzej M Kierzek. STOCKS: STOChastic Kinetic Simulations of biochemical systems with Gillespie algorithm. *Bioinformatics*, 18(3):470–481, 2002.
- [60] David C Grainger, Douglas Hurd, Marcus Harrison, Jolyon Holdstock, and Stephen JW Busby. Studies of the distribution of Escherichia coli cAMP-receptor protein and RNA polymerase along the E. coli chromosome. *Proceedings of the National Academy of Sciences of the United States of America*, 102(49):17693–17698, 2005.
- [61] Wikimedia Commons / "Bruno in Columbus". "Dictyostelium Aggregation", 2008. Public Domain.

- [62] Michael T Laub and William F Loomis. A molecular network that produces spontaneous oscillations in excitable cells of *Dictyostelium*. *Molecular biology of the cell*, 9(12):3521–3532, 1998.
- [63] Lan Ma and Pablo A Iglesias. Quantifying robustness of biochemical network models. *BMC bioinformatics*, 3(1):38, 2002.
- [64] RF Mao, V Rubio, H Chen, L Bai, OC Mansour, and ZZ Shi. OLA1 protects cells in heat shock by stabilizing HSP70. *Cell death & disease*, 4(2):e491, 2013.
- [65] ETH Zurich Control Theory and Systems Biology Group. "Regulation of E.coli's Heat-Shock Response", 2014.
- [66] MJ Kazmierczak, M Wiedmann, and KJ Boor. Alternative Sigma Factors and Their Roles in Bacterial Virulence. *Microbiology and molecular biology reviews: MMBR*, 69(4):527–543, 2005.
- [67] Miguel Gonzalez-Guzman, Lesia Rodriguez, Laura Lorenzo-Orts, Clara Pons, Alejandro Sarrion-Perdigones, Maria A Fernandez, Marta Peirats-Llobet, Javier Forment, Maria Moreno-Alvero, Sean R Cutler, et al. Tomato PYR/PYL/RCAR abscisic acid receptors show high expression in root, differential sensitivity to the abscisic acid agonist quinabactin, and the capability to enhance plant drought resistance. *Journal of experimental botany*, page eru219, 2014.
- [68] María del Carmen Rodríguez-Gacio, Miguel A Matilla-Vázquez, and Angel J Matilla. Seed dormancy and ABA signaling. The breakthrough goes on. *Plant signaling & behavior*, 4(11):1035–1048, 2009.
- [69] Takuya Miyakawa, Yasunari Fujita, Kazuko Yamaguchi-Shinozaki, and Masaru Tanokura. Structure and function of abscisic acid receptors. *Trends in plant science*, 18(5):259–266, 2013.
- [70] Takashi Hirayama and Taishi Umezawa. The PP2C-SnRK2 complex. *The central regulator of an abscisic acid signaling pathway. Plant Signal Behav*, 5: 160–163, 2010.
- [71] LN Johnson. The regulation of protein phosphorylation. *Biochemical Society Transactions*, 37(Pt 4):627–641, 2009.
- [72] Richard S Smith. The role of auxin transport in plant patterning mechanisms. *PLoS Biology*, 6(12):e323, 2008.
- [73] Somnath Karmakar and Bholanath Mandal. Graph Theoretical Solutions for the Coupled Kinetic Rate Equations. *The Journal of Physical Chemistry A*, 118(7):1155–1161, 2014.
- [74] Marco Lopez-Ilasaca, Xiushi Liu, Koichi Tamura, and Victor J Dzau. The angiotensin II type I receptor-associated protein, ATRAP, is a transmembrane protein and a modulator of angiotensin II signaling. *Molecular biology of the cell*, 14(12):5038–5050, 2003.

- [75] B Alberts et al. *Molecular Biology of the Cell*. 4th Ed. Garland Science, 2002.
- [76] Björn H Falkenburger, Eamonn J Dickson, and Bertil Hille. Quantitative properties and receptor reserve of the DAG and PKC branch of Gq-coupled receptor signaling. *The Journal of general physiology*, 141(5):537–555, 2013.
- [77] John Conway. The Game of Life. *Scientific American*, 223(4):4, 1970.
- [78] Andrzej M Kierzek, Jolanta Zaim, and Piotr Zielenkiewicz. The effect of transcription and translation initiation frequencies on the stochastic fluctuations in prokaryotic gene expression. *Journal of Biological Chemistry*, 276(11):8165–8172, 2001.
- [79] William S Klug, Michael R Cummings, et al. *Concepts of Genetics*. Number Ed. 8. Pearson Education, Inc, 2005.
- [80] Robert C Brewster, Daniel L Jones, and Rob Phillips. Tuning promoter strength through RNA polymerase binding site design in *Escherichia coli*. *PLoS computational biology*, 8(12):e1002811, 2012.
- [81] Soumen Roy. Networks, Metrics, and Systems Biology. In *A Systems Theoretic Approach to Systems and Synthetic Biology I: Models and System Characterizations*, pages 211–225. Springer Netherlands, 2014. ISBN 978-94-017-9040-6.
- [82] Tero Aittokallio and Benno Schwikowski. Graph-based methods for analysing networks in cell biology. *Briefings in Bioinformatics*, 7(3):243–255, 2006.
- [83] Herbert M Sauro, Michael Hucka, Andrew Finney, Cameron Wellock, Hamid Bolouri, John Doyle, and Hiroaki Kitano. Next generation simulation tools: the Systems Biology Workbench and BioSPICE integration. *Omics A Journal of Integrative Biology*, 7(4):355–372, 2003.
- [84] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S Baliga, Jonathan T Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome research*, 13(11):2498–2504, 2003.
- [85] Chen Li, Marco Donizelli, Nicolas Rodriguez, Harish Dharuri, Lukas Endler, Vijayalakshmi Chelliah, Lu Li, Enuo He, Arnaud Henry, Melanie I. Stefan, Jacky L. Snoep, Michael Hucka, Nicolas Le Novère, and Camille Laibe. BioModels Database: An enhanced, curated and annotated resource for published quantitative kinetic models. *BMC Systems Biology*, 4:92, Jun 2010.
- [86] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal, Complex Systems*:1695, 2006.
- [87] John Hopcroft and Robert Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.
- [88] Tore Opsahl, Filip Agneessens, and John Skvoretz. Node centrality in weighted networks: Generalizing degree and shortest paths. *Social Networks*, 32(3):245–251, 2010.

- [89] Linton C Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1979.
- [90] Mark EJ Newman. Scientific collaboration networks. ii. shortest paths, weighted networks, and centrality. *Physical review E*, 64(1):016132, 2001.
- [91] Edsger W Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische mathematik*, 1(1):269–271, 1959.
- [92] Linton C Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, pages 35–41, 1977.
- [93] Soumen Roy. Systems biology beyond degree, hubs and scale-free networks: the case for multiple metrics in complex networks. *Systems and Synthetic Biology*, 6:31–34, 2012. 10.1007/s11693-012-9094-y.
- [94] Ulrik Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.
- [95] Mohsen Bayati, Andrea Montanari, and Amin Saberi. Generating Random Graphs with Large Girth. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 566–575. Society for Industrial and Applied Mathematics, 2009.
- [96] Thomas Schank and Dorothea Wagner. Approximating clustering coefficient and transitivity. *Journal of Graph Algorithms and Applications*, 9:2005, 2005.
- [97] Duncan J Watts and Steven H Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440–442, 1998.
- [98] Douglas R White and Frank Harary. The Cohesiveness of Blocks in Social Networks: Node Connectivity and Conditional Density. *Sociological Methodology*, 31(1):305–359, 2001.
- [99] Abdol-Hossein Esfahanian. Connectivity algorithms, 2010.
- [100] Vladimir Filkov, Zachary M Saul, Soumen Roy, Raissa M D’Souza, and Premkumar T Devanbu. Modeling and verifying a broad array of network properties. *EPL (Europhysics Letters)*, 86(2):28003, 2009.
- [101] Stefan Hoops, Sven Sahle, Ralph Gauges, Christine Lee, Jürgen Pahle, Natalia Simus, Mudita Singhal, Liang Xu, Pedro Mendes, and Ursula Kummer. COPASI—a COMplex PATHway Simulator. *Bioinformatics*, 22(24):3067–3074, 2006.
- [102] Kevin R. Sanft, Sheng Wu, Min Roh, Jin Fu, Rone Kwei Lim, and Linda R. Petzold. StochKit2: software for discrete stochastic simulation of biochemical systems with events. *Bioinformatics*, 27(17):2457–2458, 2011.
- [103] Thomas W. Evans, Colin S. Gillespie, and Darren J. Wilkinson. The SBML discrete stochastic models test suite. *Bioinformatics*, 24(2):285–286, 2008.

- [104] Rajesh Ramaswamy, Nelido Gonzalez-Segredo, and Ivo F. Sbalzarini. Partial-Propensity Direct Method - Notes and Corrections. http://mosaic.mpi-cbg.de/docs/PDM_NotesAndCorrections.pdf. Accessed: 2015-01-24.
- [105] David Roxbee Cox and Hilton David Miller. *The Theory of Stochastic Processes*. 1965.
- [106] Sean Mauch and Mark Stalzer. Efficient Formulations for Exact Stochastic Simulation of Chemical Systems. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(1):27–35, 2011.
- [107] Deepak Chandran, Frank Bergmann, and Herbert Sauro. TinkerCell: modular CAD tool for synthetic biology. *Journal of Biological Engineering*, 3(1):19, 2009.
- [108] Vladimir N. Vapnik. *The nature of statistical learning theory*. Springer-Verlag New York, New York, NY, USA, 1995. ISBN 0-387-94559-8.
- [109] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A Library for Large Linear Classification. *Journal of Machine Learning Research*, 9:1871–1874, June 2008.
- [110] Hsiang-Fu Yu, Fang-Lan Huang, and Chih-Jen Lin. Dual coordinate descent methods for logistic regression and maximum entropy models. *Machine Learning*, 85(1-2):41–75, October 2011.
- [111] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1–27:27, May 2011.
- [112] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey McLachlan, Angus Ng, Bing Liu, Philip Yu, Zhi-Hua Zhou, Michael Steinbach, David Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.
- [113] Mark A Hall. *Correlation-based Feature Selection for Machine Learning*. PhD thesis, The University of Waikato, 1999.
- [114] Damjan Krstajic, Ljubomir J Buturovic, David E Leahy, and Simon Thomas. Cross-validation pitfalls when selecting and assessing regression and classification models. *Journal of Cheminformatics*, 6(1):1–15, 2014.
- [115] Paul H Lee. Resampling Methods Improve the Predictive Power of Modeling in Class-Imbalanced Datasets. *International Journal of Environmental Research and Public Health*, 11(9):9776–9789, 2014.
- [116] Mikel Galar, Alberto Fernandez, Eudene Barrenechea, Humberto Bustince, and Francisco Herrera. A Review on Ensembles for the Class Imbalance Problem: Bagging, Boosting, and Hybrid-Based Approaches. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(4):463–484, 2012.

- [117] P.W. Katz. String searcher, and compressor using same, 1991. US Patent 5,051,745.
- [118] Michel F Sanner et al. Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61, 1999.
- [119] Massimo Di Pierro. Web2Py for Scientific Applications. *Computing in Science and Engg.*, 13(2):64–69, March 2011.
- [120] Glenn E Krasner, Stephen T Pope, et al. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Journal of object oriented programming*, 1(3):26–49, 1988.
- [121] Mike Owens. *SQLite*. Springer, 2010.
- [122] Chuck Musciano and Bill Kennedy. *HTML*. O'Reilly & Associates, 1997.
- [123] Luka Cvrk. "watchthis!" Web Theme, 2008. Creative Commons.
- [124] Jon Duckett. *JavaScript and JQuery: Interactive Front-End Web Development*. Wiley Publishing, 2014.
- [125] License, GNU General Public. Version 3. *Free Software Foundation*. <http://www.gnu.org/copyleft/gpl.html>, 2007.
- [126] Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education, 1986.
- [127] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [128] Marcin Kalicinski. *RapidXml*, 2009.
- [129] Richard M Stallman. *Using and Porting the GNU Compiler Collection*, volume 86. Free Software Foundation, 1999.
- [130] Axel Naumann. Preparing for the new C++ 11 standard. In *Journal of Physics: Conference Series*, volume 396, page 052056. IOP Publishing, 2012.
- [131] Martin Wojtczyk and Alois Knoll. A Cross Platform Development Workflow for C/C++ Applications. In *Software Engineering Advances, 2008. ICSEA'08. The Third International Conference on*, pages 224–229. IEEE, 2008.
- [132] C++ Boost Libraries. <http://www.boost.org>, 2008.
- [133] David R Musser, Gillmer J Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley Professional, 2009.

- [134] Brian Gough. *GNU Scientific Library Reference Manual*. Network Theory Ltd., 2009.
- [135] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1): 3–30, 1998.
- [136] Mutsuo Saito and Makoto Matsumoto. SIMD-oriented fast Mersenne Twister: a 128-bit pseudorandom number generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer, 2008.
- [137] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An Overview of the HDF5 Technology Suite and its Applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.
- [138] Open Source Initiative et al. The BSD license, 2006.
- [139] J Hill, L Smith, and A Trew. Profiling Write Performance of HDF5. Technical report, EPCC, University of Edinburgh, JCMB, Kings Buildings, Edinburgh, EH9 3JZ, 2005.
- [140] Marcin Kalicinski. RapidXml Manual.
- [141] Hiroyuki Kuwahara and Ivan Mura. An Efficient and Exact Stochastic Simulation Method to Analyze Rare Events in Biochemical Systems. *The Journal of chemical physics*, 129(16):165101, 2008.
- [142] Timothy J Rudge, Paul J Steiner, Andrew Phillips, and Jim Haseloff. Computational Modeling of Synthetic Microbial Biofilms. *ACS Synthetic Biology*, 1(8): 345–352, 2012.
- [143] Angel Goñi-Moreno, Martyn Amos, and Fernando de la Cruz. Multicellular Computing Using Conjugation for Wiring. *PLoS ONE*, 8(6), 2013.
- [144] Antonio Prestes García and Alfonso Rodríguez-Patón. BactoSim—An Individual-Based Simulation Environment for Bacterial Conjugation. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Sustainability: The PAAMS Collection*, pages 275–279. Springer International Publishing, 2015.
- [145] Sorana-Daniela Bolboaca and Lorentz Jäntschi. Pearson versus Spearman, Kendall’s tau correlation analysis on structure-activity relationships of biologic active compounds. *Leonardo Journal of Sciences*, 9:179–200, 2006.
- [146] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

- [147] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, pages 80–83, 1945.
- [148] William H Kruskal and W Allen Wallis. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952.
- [149] Richard Lowry. *VassarStats: Website for Statistical Computation*. Vassar College, 2004.
- [150] Samuel Sanford Shapiro and Martin B Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, pages 591–611, 1965.
- [151] Samuel S Shapiro, Martin B Wilk, and Hwei J Chen. A Comparative Study of Various Tests for Normality. *Journal of the American Statistical Association*, 63(324):1343–1372, 1968.
- [152] JP Royston. An Extension of Shapiro and Wilk's W Test for Normality to Large Samples. *Applied Statistics*, pages 115–124, 1982.
- [153] Patrick Royston. Approximating the Shapiro-Wilk W-Test for non-normality. *Statistics and Computing*, 2(3):117–119, 1992.
- [154] Ross Ihaka and Robert Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.
- [155] Philipp K Janert. *Data Analysis with Open Source Tools*. "O'Reilly Media, Inc.", 2010.
- [156] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Machine learning*, 20(3):273–297, 1995.
- [157] Wikimedia Commons ('Cyc'). "Graphic showing the maximum separating hyperplane and the margin.", 2008. Public Domain.
- [158] William H Press. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007.
- [159] John H McDonald. *Handbook of Biological Statistics*. Sparky House Publishing, 2009.
- [160] T. Cover and P. Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, January 1967.
- [161] David W Aha, Dennis Kibler, and Marc K Albert. Instance-Based Learning Algorithms. *Machine learning*, 6(1):37–66, 1991.
- [162] Pádraig Cunningham and Sarah Jane Delany. k-Nearest Neighbour Classifiers. 2007.
- [163] Ron Kohavi et al. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAI*, volume 14, pages 1137–1145, 1995.